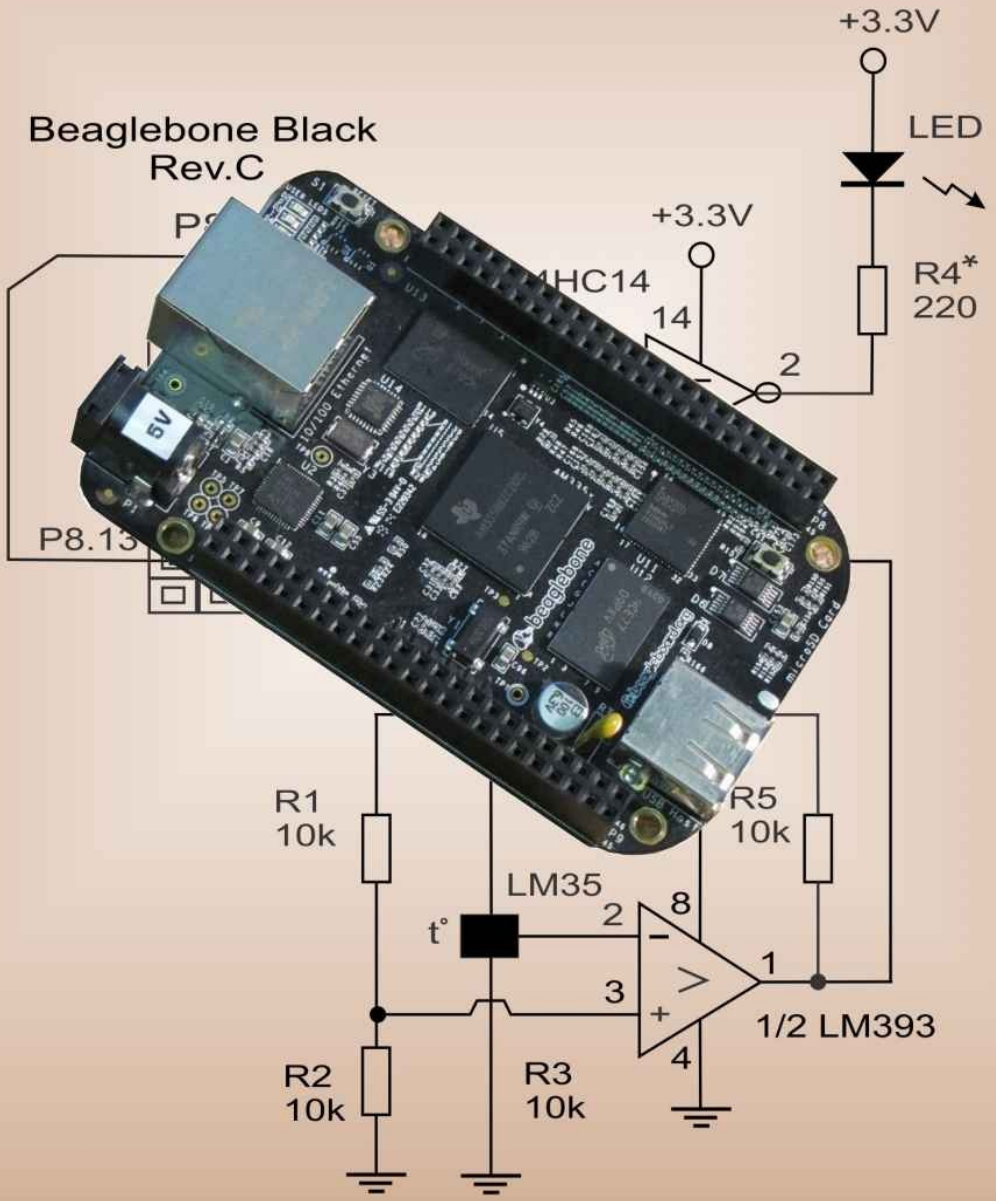
Yury Magda

BeagleBone Black Interfacing

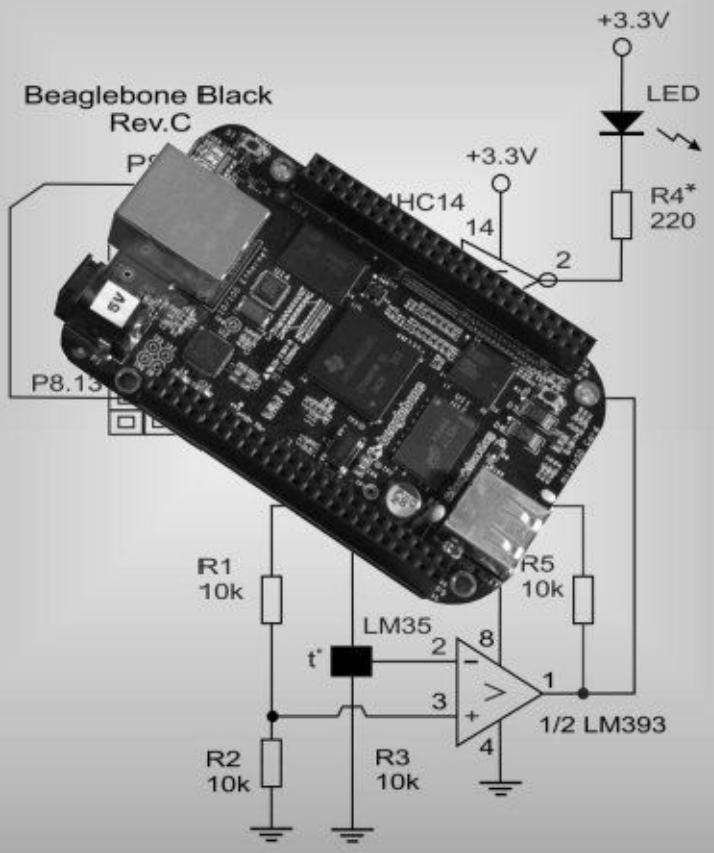
hardware and software



Yury Magda

BeagleBone Black Interfacing

hardware and software



BeagleBone Black Interfacing

Hardware and Software

By Yury Magda

The programs, examples, and applications presented in this book have been included for their instructional value. The author offer no warranty implied or express, including but not limited to implied warranties of fitness or merchantability for any particular purpose and do not accept any liability for any loss or damage arising from the use of any information in this book, or any error or omission in such information, or any incorrect use of these programs, procedures, and applications.

Contents

Introduction

The popular BeagleBone Black miniature computer besides being used as a low-power general-purpose desktop computer can form a basis for building measurement and control systems by both professionals and hobbyists. This is due to the general purpose input/output port (GPIO) and analog channels available on the BeagleBone Black board; each GPIO pin or analog channel can be accessed through the P8 or P9 headers.

This book describes projects which may be used as templates for building various measurement and control systems. Each project includes both hardware and software accompanied by the detail description of what is doing.

Note that Debian OS is a true Linux operating system, so developers cannot develop applications for measuring and control system operating in ″ real-time ″ without writing or using specific kernel drivers. Nevertheless, it is possible to develop numerous applications that don’t require fast responses and short time intervals. Using an additional library developed for Python allows to measure relatively slow analog (continuous) and digital signals coming from various (temperature, humidity, pressure, light, etc.) sensors. The programs written in Python can also drive external loads (motors, relays) by bringing digital signals on GPIO pins.

Part of the material is dedicated to use of the BeagleBone Black in network measurement and control systems. A few projects illustrate building the network systems where the BeagleBone Black is used together with the popular Tiva™ C Series Connected LaunchPad board from Texas Instruments. The BeagleBone Black can be easily fitted into wireless systems using the Wixel Programmable USB Wireless module from Pololu Corp., so the readers can explore a couple of projects presented in this guide. Several projects illustrate the design of wireless measurement and control systems using a popular Bluetooth interface.

This book is thought as a highly practical guide for building measurement and control systems. The material of the book assumes that the readers are familiar, at least, with basics of designing and assembling electronic circuits. For most projects having some basic skills in electronics will serve the readers well and allow them to understand what is going on behind the scenes. Each project is accompanied by the brief description which helps to make things clear.

All projects were designed using the BeagleBone Black Rev.C board running Debian

Linux. All of the source code for the BeagleBone projects was developed in Python. A few projects used Arduino Uno and Arduino Due development boards. Most projects described in this guide can be easily improved or modified if necessary.

Disclaimer

The design techniques described in this book have been tested on the BeagleBone Black Rev.C board without damage of the equipment. I will not accept any responsibility for damages of any kind due to actions taken by you after reading this book.

BeagleBone Black GPIO pins

Development platforms like BeagleBone provide an environment for software and hardware experimenting. This is possible because availability of General Purpose Input/Output pins (GPIO) on the BeagleBone Black board.

The BeagleBone Black is equipped with the two sets of headers labeled P8 and P9. These headers run along the edges of the board. Each of them has 46 pins with pins 1, 2, 45, and 46 labeled on each header. To identify pin numbers in the middle, you'll have to count off from the pins on the end.

The GPIO pins can be employed in various control and measurement systems for reading sensors, communicating with other electronics, and much more. Most pins can operate in different modes thus accommodating different possible functions. Note that digital pins must be either high or low.

Basically, GPIO pins may be put in input or output mode. When output, a pin can be pulled either low (log."0") or high (log."1") by the program code; this allows to control external circuitry connected to this pin. The logical level "0" corresponds to the voltage level close to 0 volts, while the log."1" is about 3.3 volts.

Reading the state of a loose pin returns the unpredictable result. Each GPIO pin can individually be configured either as input or output. All the GPIO pins can also be reconfigured to provide alternate functions, SPI, PWM, I^2C and so.

Note that GPIO voltage levels are 3.3 V and are not 5 V tolerant. The 3.3 V logic level of the BeagleBone Black assumes that a developer has to use only 3.3 volt logic components when connecting an external circuitry.

The BeagleBone Black board has no over-voltage protection – the intention is that people interested in serious interfacing will use an external board with buffers, level conversion and analog I/O rather than soldering directly onto the main board.

Chances are that external circuitry could be poorly assembled and have short circuits, so I highly recommend you to employ a standalone DC 5V/3.3V power supply for powering your electronic circuits to prevent the permanent damage of the BeagleBone Black.

Another aspect of interfacing the BeagleBone is maximum allowable current through an GPIO pin. The current has to be less than 8 mA, so any load that draws more current can damage the board. The best solution when connecting external circuits to the BeagleBone Black is to employ some CMOS buffer IC (for example, 74HC00, 74HC08, 74HC14, etc.) which isolates the external circuitry from low-current GPIO

pins.

Be very careful when wiring the GPIO pins – before probing P8 or P9 pins, it's a good idea to strip short pieces of insulation off a wire and push them over the 3.3V pins so you don't accidentally short them with a probe.

When attaching your electronic circuits to the BeagleBone GPIO pins use short wires (10-12cm long) so that to ensure minimum distortion of digital signals. Using long wires may cause your circuit to stop functioning properly.

Powering external circuits

The low-power external circuitry driven by the BeagleBone Black can be powered using power supply +3.3V located on the board. This, however, poses a risk of damaging the BeagleBone board if external circuitry comprises a short circuit. For that reason, it would better to power such circuits from a stand-alone DC voltage source.

When both +3.3V and +5V voltage are needed, we can take some DC power supply +5V and connect its output to a low-dropout regulator (LDO) chip providing the steady +3.3V output. The following power circuit was used for powering all projects from this book (**Fig.1**).

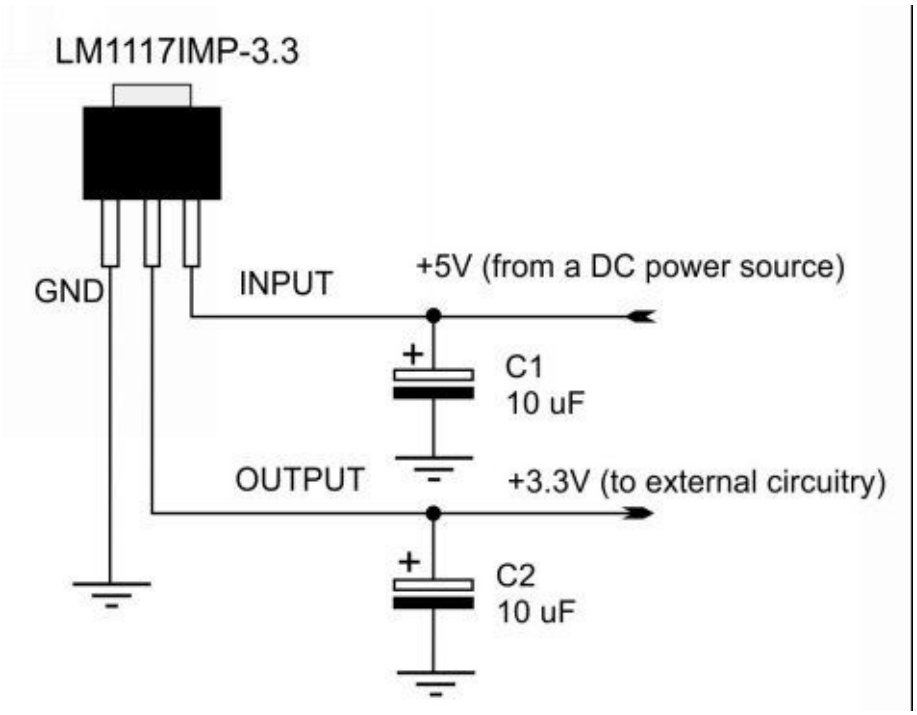


Fig.1

Here the +3.3V DC voltage is obtained from +5V supply using the LDO chip LM1117-3.3. The current flowing through external circuitry shouldn’t exceed the maximum provided by LDO (800 mA for the LM1117). Obviously, any other LDO chip with +3.3V output and suitable rated current can be applied as well.

Configuring software

All projects in this guide will use the Python language. Since we will mainly deal with GPIO pins, we can apply the Adafruit BeagleBone IO Python Library. That is a set of Python tools to allow GPIO, PWM, and ADC access on the BeagleBone using the Linux 3.8 Kernel and above.

Note that this library may changes as development continues, so it would be worth to read the release notes each time you update the library. Since our BeagleBone runs Debian the following sequence can be executed to install Adafruit IO Python library.

```
$ sudo ntpdate pool.ntp.org
$ sudo apt-get update
$ sudo apt-get install build-essential python-dev python-pip -y
$ sudo pip install Adafruit_BBIO
```

More complex procedure can be performed manually:

```
$ sudo ntpdate pool.ntp.org
$ sudo apt-get update
$ sudo apt-get install build-essential python-dev python-pip -y
$ git clone git://github.com/adafruit/adafruit-beaglebone-io-python.git
$ cd adafruit-beaglebone-io-python
$ sudo python setup.py install
$ cd ..
$ sudo rm -rf adafruit-beaglebone-io-python
```

Once the IO library has been installed, we can test it by launching several examples below. The following sequence illustrates configuring the GPIO pin **P8.14** as output.

```
import Adafruit_BBIO.GPIO as GPIO
GPIO.setup(“P8_14”, GPIO.OUT)
GPIO.output(“P8_14”, GPIO.HIGH)
```

To configure the **P8.14** GPIO pin as input we can apply the following:


```
GPIO.setup("P8_14", GPIO.IN)
```

When operating as input, the **P8.14** pin can be checked by the following sequence:

```
if GPIO.input("P8_14"):
    print("HIGH")
else:
    print("LOW")
```

We can also check when the falling, rising edge or both arrives at an input pin. This is possible by calling one of the functions

wait_for_edge

event_detected

These functions can take either of GPIO.RISING, GPIO.FALLING, or GPIO.BOTH constants as their parameter. The following statement

```
GPIO.wait_for_edge("P8_14", GPIO.RISING)
```

waits until the rising edge arrives on the **P8.14** input.

If waiting is impossible, we can simply check if one of the events (rising edge, falling edge or both) has occurred. The following sequence checks when pin **P9.12** is brought from high to low (falling edge):

```
GPIO.add_event_detect("P9_12", GPIO.FALLING)
…
…
…
```

```
if GPIO.event_detected(“P9_12”):
    print “event detected!”
```

Adafruit IO library also allows to configure PWM channels of the BeagleBone Black using the PWM module. The functions from this module allow to obtain the pulse train with variable frequency, duty and polarity. The next example illustrates this:

```
import Adafruit_BBIO.PWM as PWM
import time

duty = float(input(‘Enter the duty cycle (0-100):’))
freq = input(‘Enter the frequency, Hz: ‘)



PWM.start(“P8_13”, duty, freq, 0)
time.sleep(30)

PWM.stop(“P8_13”)
PWM.cleanup()
```

Note that duty values can range between 0 (off) and 100 (on).

This program produces the following output at different values of the frequency and duty cycle:

```
$ sudo python Simple_PWM.py
Enter the duty cycle (0-100):23.5
Enter the frequency, Hz: 5270
$ sudo python Simple_PWM.py
Enter the duty cycle (0-100):85
Enter the frequency, Hz: 5000
$ sudo python Simple_PWM.py
Enter the duty cycle (0-100):73.7
```

Enter the frequency, Hz: 4900

We can also set frequency, duty or polarity by using the following functions (pin **P9.14** is involved):

```
PWM.start("P9_14", 50)
PWM.set_duty_cycle("P9_14", 32.5)
PWM.set_frequency("P9_14", 100)

PWM.stop("P9_14")
PWM.cleanup()

#set polarity to 1 on start:
PWM.start("P9_14", 50, 2000, 1)
```

The ADC module includes the functions allowing to process analog signals by a built-in Analog-to-Digital converter. This is illustrated by the next example:

```
import Adafruit_BBIO.ADC as ADC
ADC.setup()

#the read() function returns values in the range of 0 to 1.0
value = ADC.read("P9_40")

#the read_raw() function returns non-normalized value
value = ADC.read_raw("P9_40")
```

Processing analog signals will be discussed in detail a bit later.

Connecting GPIO outputs to DC loads

A GPIO pin cannot draw a heavy load since its output current is limited to a few milliamperes (≈8mA). Most external circuits, however, draw much more current, so the GPIO pin should be followed by some buffer isolating a microprocessor outputs from a load.

Anyway, it’s bad practice to connect any, even a low power load directly to microprocessor pins. Occasionally, even an ordinary LED may turn out to be short thus overloading a GPIO pin. Using a direct connection of GPIO pins to loads may also cause overheating the microprocessor because of consuming too much power. This can lead to gradual degradation and destruction of the chip.

Buffer outputs themselves can draw relatively high current, so some stand-alone DC power supply should be applied to feed them. A few buffer circuits are presented below (**Fig.2** – **Fig.5**).

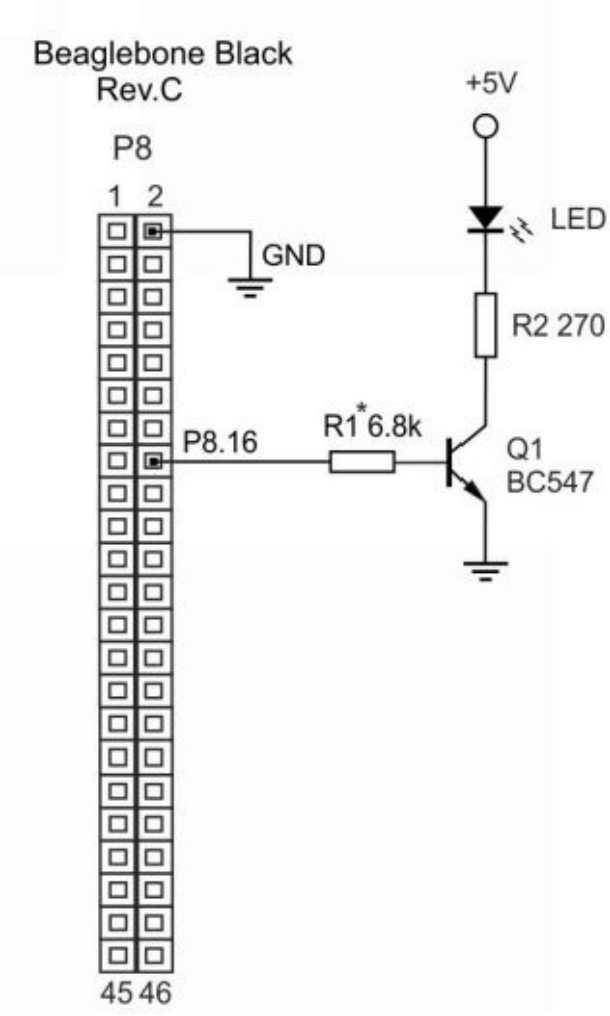


Fig.2

In the above circuit (**Fig.2)**, the external load (LED) is driven by pin **P8.16** via the small-signal bipolar transistor Q1. The Q1 device acts as a switch which opens when the **P8.16** pin is pulled high (log."1", close to 3.3V) and closes when **P8.16** is pulled low (log."0", about 0 V). This configuration is suitable for loads drawing the current ranging from a few to tens milliamperes. Almost any small-signal general purpose signal bipolar transistor will fit this circuit (2N2222, 2N4904, 2N4401, etc.). The value of R1 may be 5-10k.

When a load consumes a relatively high current, a developer should take some power bipolar transistor with a large current gain (β). The good choice may be to take some Darlington device, for example, TIP122, TIP142, etc.

The buffer stage can also be arranged around a power MOSFET device (**Fig.3**). In this circuit, the power MOSFET IRF510 is used, although almost any of similar devices, for instance, IRF520, IRFZ24, etc., can be used as well. Note that general purpose power MOSFETs can reliable operate at relatively low frequencies, up to a few KHz.

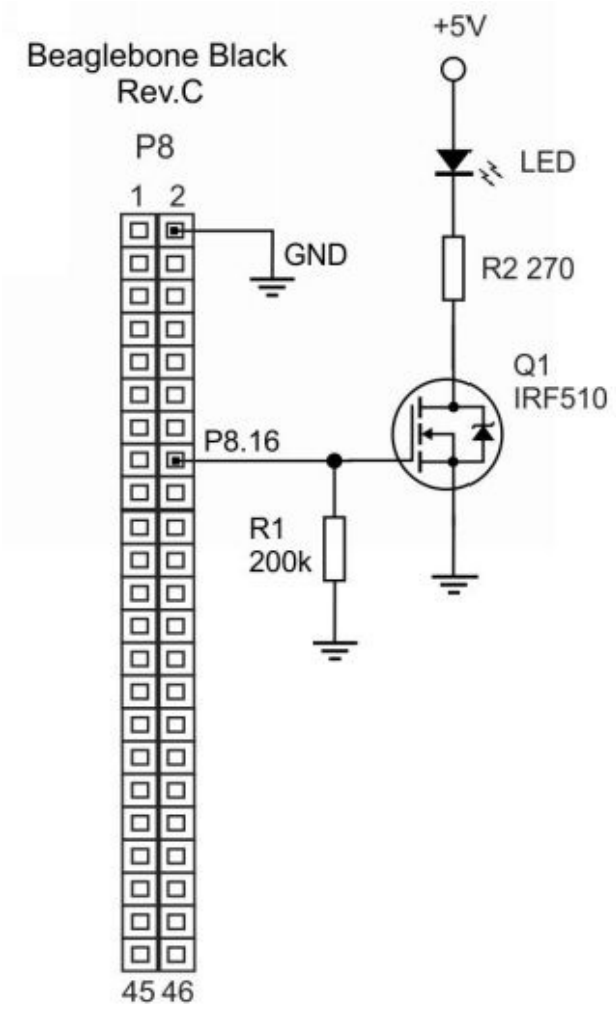


Fig.3

A special case is an inductive load (relay, motor, etc.). Dealing with such type of a circuit requires the protection of a control device. The simplest possible configuration will be a general purpose switching diode capable of passing a high peak current. **Fig.4** illustrates this concept for inductive loads powered by DC 5-12V voltage sources.

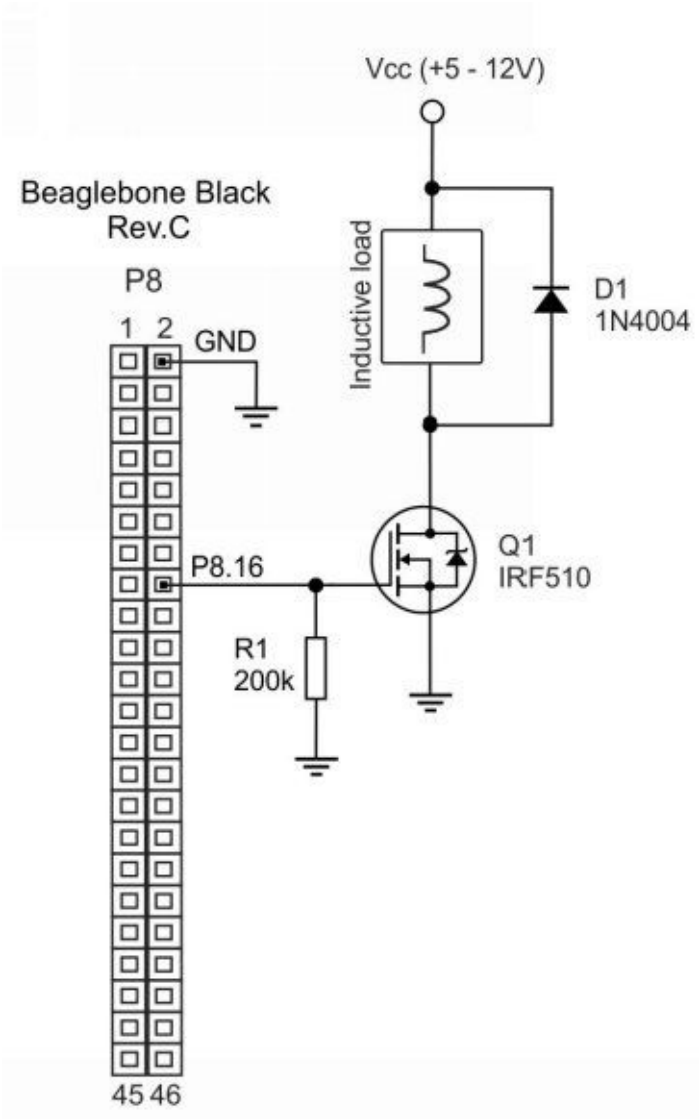


Fig.4

Alternatively, inductive loads can be driven by power bipolar transistors. The simple possible circuit is shown in **Fig.5**.

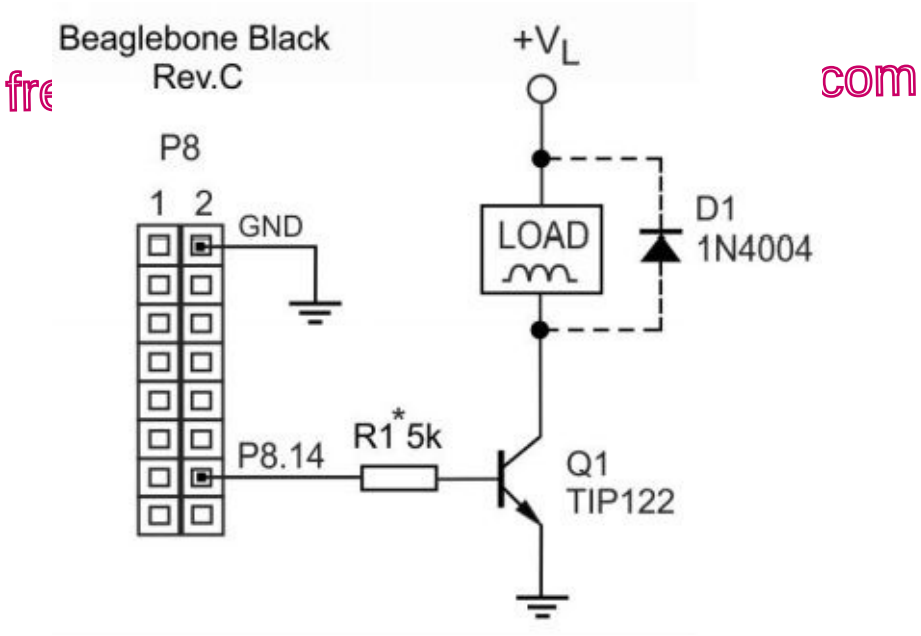


Fig.5

In this circuit, the buffer stage is arranged as the power switch using the Darlington transistor. Q1 (TIP122) is driven ON when the **P8.14** output is pulled high. This connects the DC power supply $+V_L$ to the LOAD through the open transistor Q1. Otherwise, as **P8.14**is pulled low (log. ″ 0 ″), Q1 gets cut off, thereby breaking the power chain to the LOAD. The diode D1 (noted in the dashed line) protects the power transistor Q1 while the ON-OFF transition.

This circuit works quite well, although some degree of noise may appear on the ″ ground ″ wire of the circuit while toggling inductive loads. This can be ignored if there are no sensitive conditioning circuits such as analog-to-digital converters.

The best way to prevent the high current flow through the common wire and lessen noise is to isolate low-power circuits on the BeagleBone Black side from high–power loads. This can be achieved by using the optoisolated circuits.

One possible configuration uses an optoisolator 4N35 (4N36, 4N37 can be taken as well). The circuit diagram is shown in **Fig.6**.

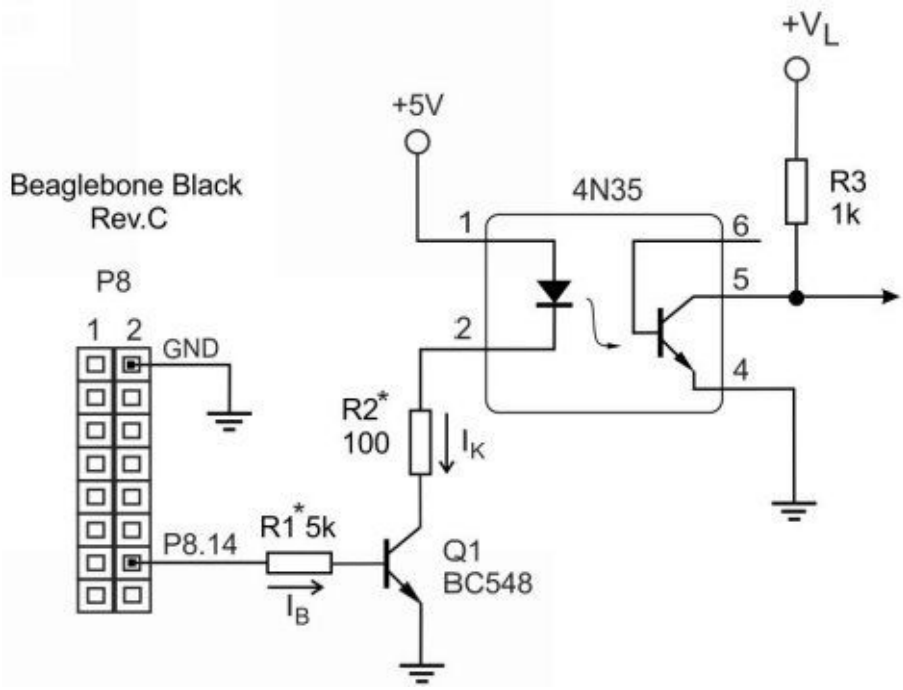


Fig.6

In this circuit the IR LED of the 4N35 optoisolator is driven ON/OFF by the bipolar transistor Q1 (BC548). The Q1 itself is controlled through the base from the **P8.14** pin configured as output. The base current I_B (in mA) will approximately be calculated as follows:

$(3.3 – 0.6) / R1$

where R1 is taken in KOhms. The base current I_B can be taken very small, so the GPIO pin will not be overloaded. Knowing the typical β of Q1, it is easily to calculate the collector current I_K flowing through the LED of the optoisolator. That gives us:

$I_K \approx \beta \times I_B$

The 4N35 – 4N37 devices are relatively slow and can't reliably operate at higher frequencies. For the frequencies of a few tens KHz and higher we need to employ high speed optoisolators. These may be 6N137, HCPL2601, etc. The optoisolator circuit with 6N3137 (f = 10 MBd) is shown in **Fig.7**.

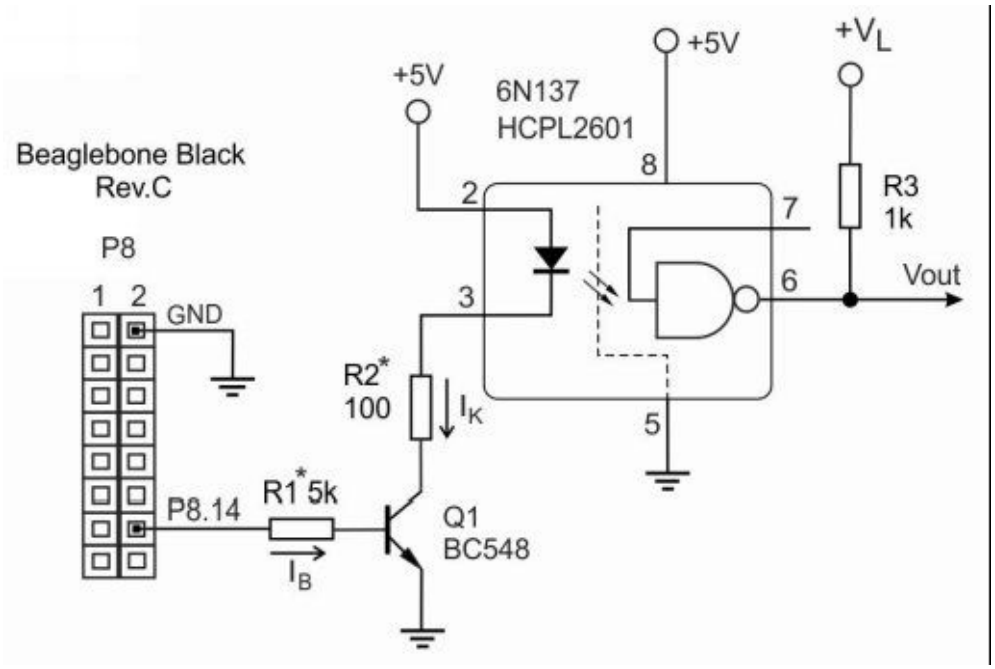


Fig.7

These optoisolators provide an open collector (OK) output which requires the pull-up resistor (R3) to be connected to pin 6. 6N137 and similar high-speed chips also provide TTL-compatible output signals suitable for processing in digital circuits.

For driving high-current DC loads we can connect an additional power switch to the output of an optoisolator. This may be, for example, a power MOSFET as is shown in **Fig.8**.

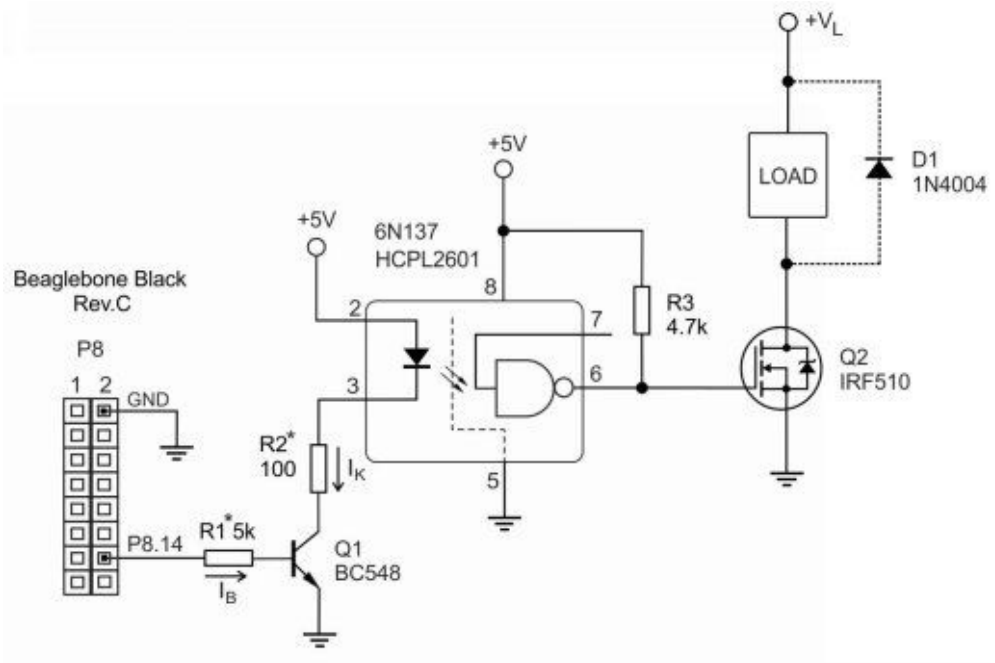


Fig.8

A variety of components may be used in this circuit. Q1 (BC548) may be replaced by any

small-signal bipolar transistor (BC547, BC549, 2N3904, 2N4401, etc.). The power MOSFET IRF510 (Q2) can be replaced by almost any power device (IRF530, IRFZ24, etc.) with suitable parameters (drain current, gate-source and drain-source voltages).

When choosing a power MOSFET, take into account the maximum power dissipation (this depends on a load and the pulse width of an input signal).

Connecting GPIO outputs to AC loads

AC loads can be driven using mechanical or solid state relays (SSR). The latter is preferable because of their reliability and high speed. This section is dedicated to interfacing zero-crossing solid state relays to AC loads.

Those SSRs are also known as a “synchronous” solid state relays; this is the most common type of SSR found in the market today. As the name implies, the switching of such relay from a non – conducting to a conducting state occurs when the AC mains voltage reaches the zero-crossing point of the sine wave. This minimizes the surge current through a load during the first conduction cycle and helps reduce the level of conducted emissions placed on the AC mains.

An ordinary solid state relay operating with AC loads usually includes a LED circuit which is optically coupled to the power switch. The LED network can be driven by the low-power DC control circuitry; the allowable DC input voltage usually ranges from 3 to 32V.

The power switch of an zero-crossing SSR consists of triac(s) with its control circuitry. The control circuits usually comprise a zero-crossing detector ensuring the reliable switching as an AC wave crosses zero.

A typical SSR have four terminals. Terminals marked with “ **–** “ and “**+**” must be wired to a low-power DC control circuit. The terminals marked “**~**” are connected to the AC high-power circuits.

The next two figures (**Fig.9** – **Fig.10**) show typical connections for the Fotek SSR-25DA solid state relay driven by the BeagleBone Black; most SSRs can be connected in much the same ways as is shown below.

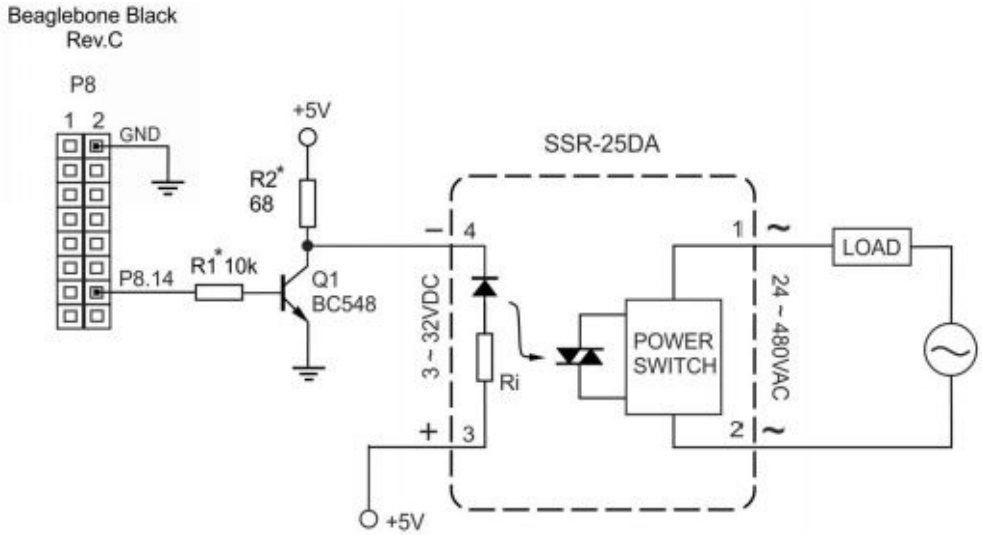


Fig.9

Fig.9 illustrates the case where the SSR is switched ON by the high level (log."1") on pin **P8.14**. When the high level is applied to the left end of the resistor R1, the transistor Q1 is driven ON and the current will flow from the +5V DC source through the terminal "+" of the SSR. The current flowing through the Ri – LED inner network causes the LED to emit visible light or infrared radiation that drives the power switch of the SSR ON.

Conversely, when pin **P8.14** is pulled low, the transistor Q1 will be cut off, so the current flow through the Ri – LED network will be blocked. This, in turn, causes the power switch to be OFF at the next zero-crossing point of an AC sine wave.

The control circuit shown in **Fig.10** drives the SSR ON by the low level on pin **P8.14**. In this case, the transistor Q1 is cut off and current flows through the LED – Ri network from +5V DC source to "ground". Conversely, the high level on pin **P8.14** opens the transistor Q1, so the current flow through LED – Ri network is cut and the power switch breaks the AC circuit.

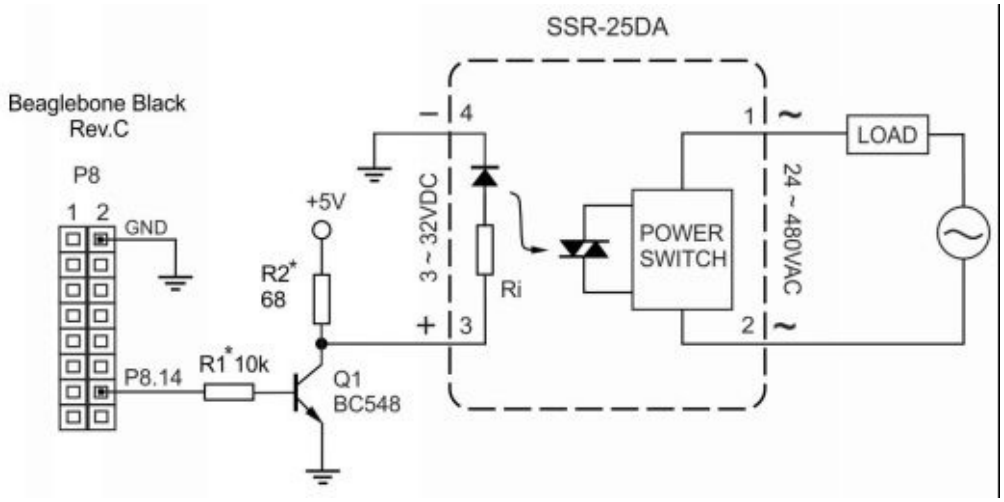


Fig.10

Almost any small-signal general purpose bipolar transistor with a maximum collector current of a few tens milliamperes will work in these circuits.

Interfacing digital input signals to the BeagleBone Black

In order to use some GPIO pin for reading digital signal, we have to configure it as input:

```
import Adafruit_BBIO.GPIO as GPIO

iPin = “P8_13”
GPIO.setup(iPin, GPIO.IN)
```

The above code puts pin **P8.13** associated with the constant **iPin** into input mode. Then the voltage level on this pin can be read by the statement:

```
iVal = GPIO.input(iPin)
```

The state of pin **P8.13** is saved in the **iVal** variable.

There are a couple of functions which simplify processing GPIO input pins. We can associate some event with a GPIO pin by introducing the function **add_event_detect()**. This function will watch out for the rising (GPIO.RISING), falling (GPIO.FALLING) or both edges (GPIO.BOTH) on the GPIO input.

Afterwards, we can check the state of a GPIO pin elsewhere in the program by invoking the **event_detected()** function. That is illustrated by the following fragment of code (**Listing 1**):

Listing 1.

```
import Adafruit_BBIO.GPIO as GPIO
import time

iPin = “P8_13”

GPIO.setup(iPin, GPIO.IN)
GPIO.add_event_detect(iPin, GPIO.BOTH)
while True:
```

```
if GPIO.event_detected(iPin):
val = GPIO.input(iPin)
if val != 0x0:
print “The RISING edge on iPin 16.”
else:
print “The FALLING edge on iPin 16.”
time.sleep(2)
```

This fragment of code determines if any event has occurred on pin **P8.13**. Once either of the watched-for events has been detected, the state of **iPin** is read by the **GPIO.input()** function.

Let’s look at how to interface input signals to the BeagleBone Black board.

The simplest case is when an input signal is taken from a mechanical switch. The connections between the switch and GPIO pin are shown in **Fig.11**.

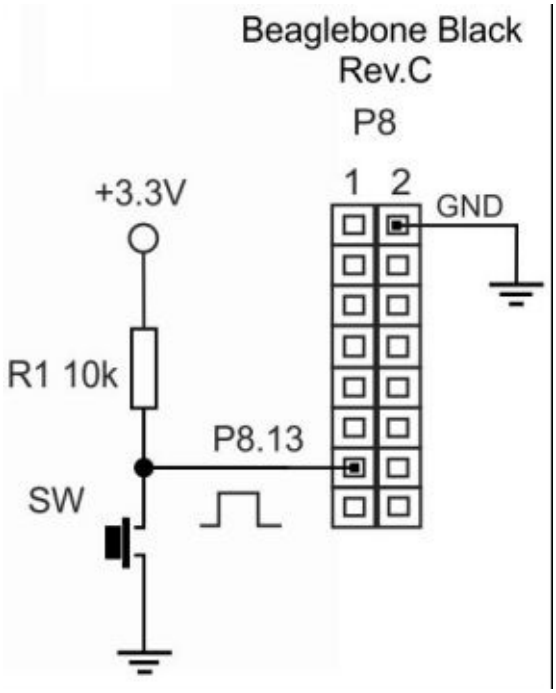


Fig.11

Assume that the GPIO pin **P8.13** is configured as input. When the switch **SW** is open, the voltage level on the **P8.13** input is high (log.”1” or 3.3V). Pressing **SW** causes the voltage level on P8.13 to drop down to 0V (log.”0”). The voltage swing “0-1” or “1-0” can be processed by software as described earlier.

But there is a problem with such simple circuit. The switch SW when toggled can cause the contact bounce on an input pin. The reason for concern is that the time it takes for contacts to stop bouncing is measured in milliseconds, while digital circuits can respond in microseconds. There are several methods to solve the problem of contact bounce (that is, to ″ de-bounce ″ the input signals). One possible approach is shown in **Fig.12**.

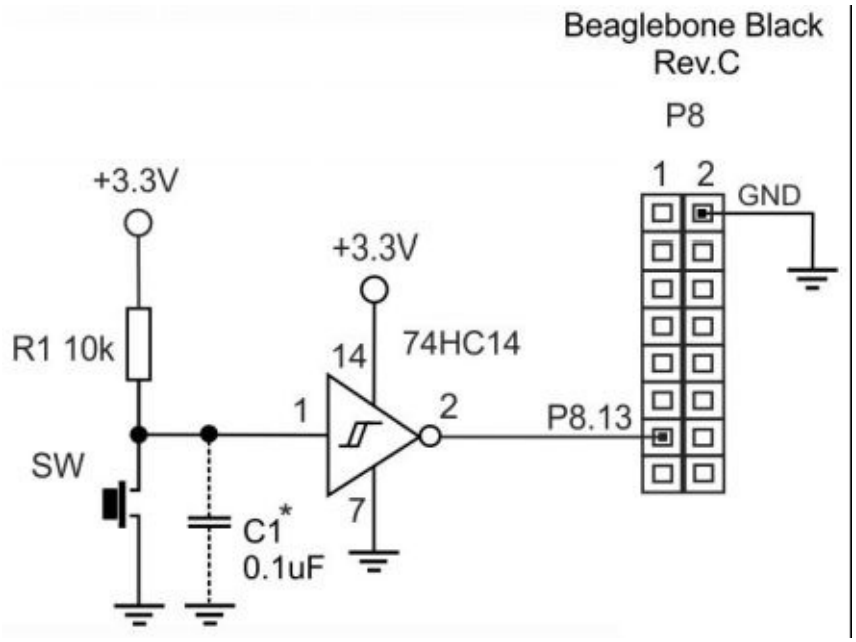


Fig.12

In this circuit, a simple hardware debounce circuit for a momentary push-button switch **SW** consists of the resistor R2 and a capacitor C1 (noted by the dashed line in **Fig.12**). Here the R2×C1 time constant is selected equal to 0.1 s (typical value) to swamp out the bounce. In general, a developer can pick R2 and C1 values to provide R2×C1 longer than the expected bounce time. The Schmitt trigger of 74HC14 connected to the switch network produces a sharp high-to-low transition. When using such debounce circuit, we should wait before pressing the switch again. If the switch has been pressed again too soon, it will not generate another signal.

Many times we need to know if any of a group of signals coming from sensors is active. In this case we can put all signals together using a circuit shown in **Fig.13**.

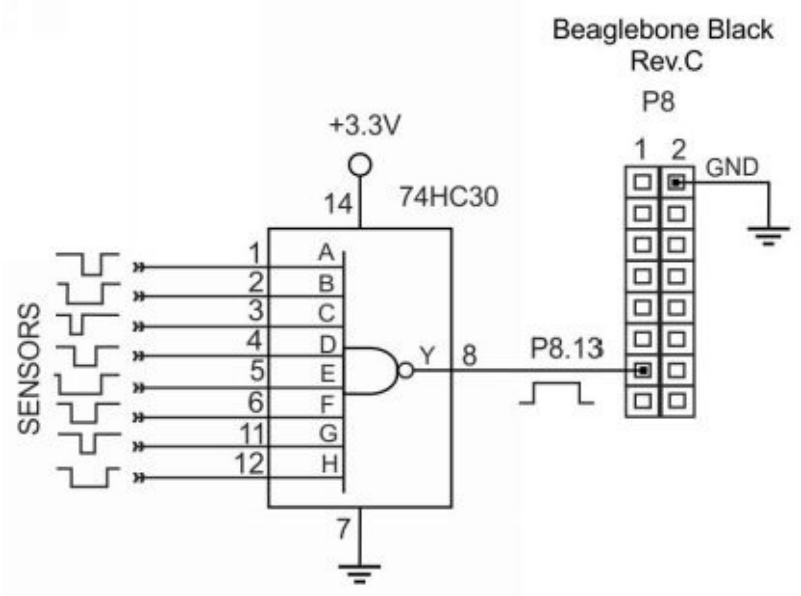


Fig.13

In this circuit, we employ an 8-input NAND gate 74HC30. If any of inputs **A** through **H** is brought low, the output **Y** goes high. Note that the active state of any sensor output should be low, so that to raise the signal on the output **Y**. If all sensor outputs are high, then the output **Y** stays low.

To obtain more input channels than the BeagleBone Black board can offer we can use a simple circuit with a multiplexer IC 74HC4051. The multiplexer allows to route multiple input signals through its single output, so the program code can pick up multiple inputs using a single GPIO pin.

The following demo project allows to read 8 digital input signals via a single digital input port of the BeagleBone Black. The schematic duagram of the project is given in **Fig.14**.

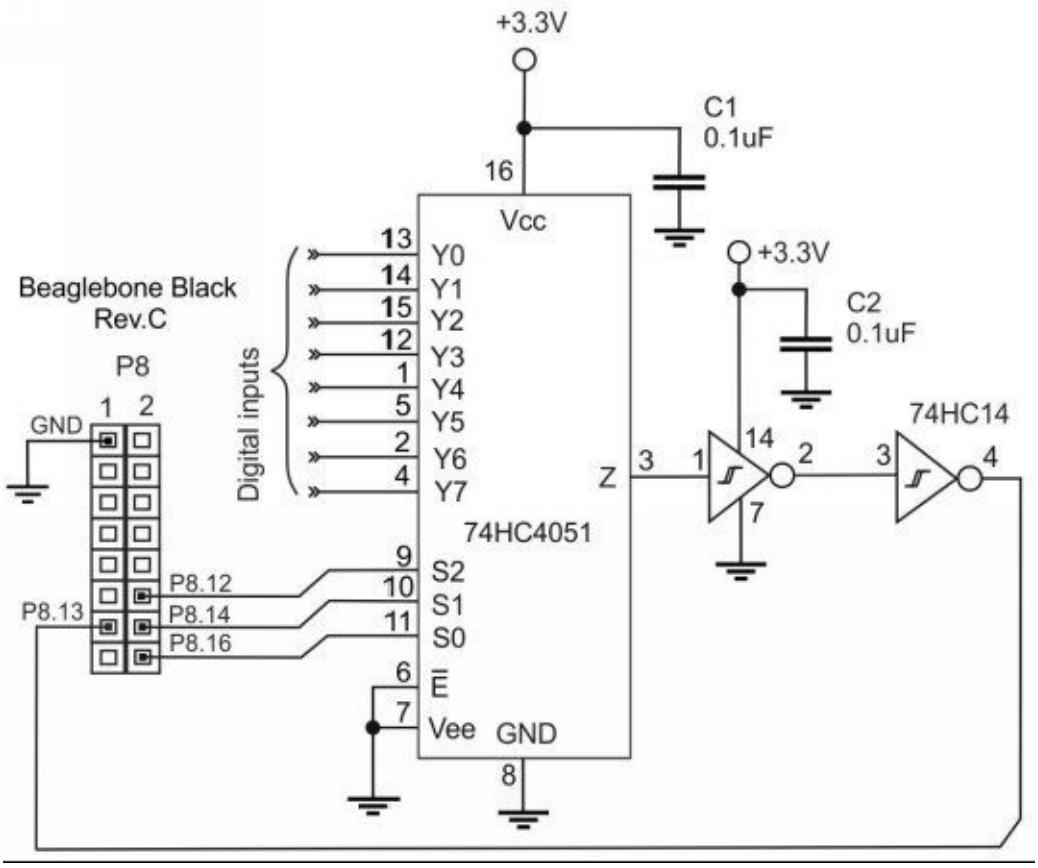


Fig.14

In this circuit, the digital signals are brought to the **Y0** – **Y7** inputs of the multiplexer 74HC4051. The number of the input channel being read (0 through 7) is determined by the binary weighted code on pins **S0** – **S2** of the multiplexer. **S0** – **S2** lines are driven by pins **P8.12**, **P8.14** and **P8.16** configured as outputs.

The signal taken from the selected input line of the multiplexer appears on the output **Z** (pin 3 of 4051). The logical level on the **Z** line is read through pin **P8.13** configured as input.

For example, in order to pick up the digital signal on the input line **Y2**, we need to put the binary code 0x2 on the lines **S0** – **S2** prior to reading the **Z** output. In this case, pins **P8.12**, **P8.14** and **P8.16** must be set as follows:

P8.12 = LOW

P8.14 = HIGH

P8.16 = LOW

For reading the input **Y5**, for example, the following combination will be valid:

P8.12 = HIGH

P8.14 = LOW

P8.16 = HIGH

The Schmitt trigger inverting buffers of 74HC14 transform slowly changing input signals into sharply defined, jitter-free output signals.

The ″ ground ″ wire of the circuit should be tied to one of the ″ ground ″ pins of the BeagleBone; the bypass capacitors C1 and C2 should be tied closely to the power pins of the chips.

The table below specifies connections between the BeagleBone Black board and the external circuitry.

BeagleBone P8 pin	**External line**
P8.13	Pin 4 of 74HC14 (signal from the Z output of the multiplexor)
P8.16	Pin 11 of 74HC4051 (S0)
P8.14	Pin 10 of 74HC4051 (S1)
P8.12	Pin 9 0f 74HC4051 (S2)

Often a system should process input pulses with various width and/or slow edges. When an input signal has a short pulse width (a few microseconds, for example), the BeagleBone Python application can’t catch such a signal and process it in a timely manner. In those cases it would be worth to lengthen the pulse width of an input signal to a few tens or hundreds microseconds so that a system would be capable of reliable processing the pulse.

This is illustrated in **Fig.15**.

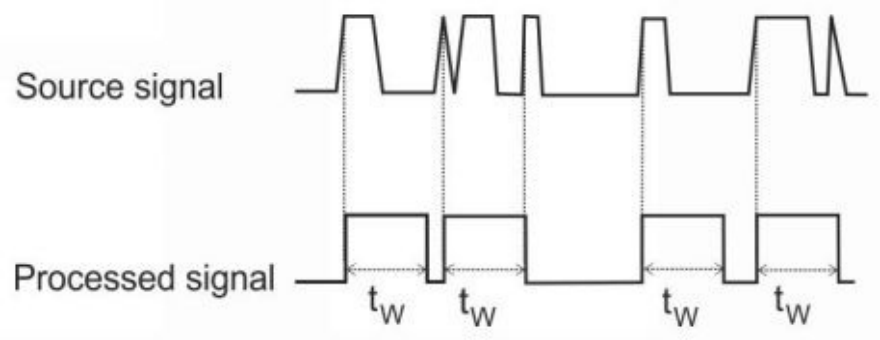


Fig.15

In the above diagram, the pulses with various widths and edges are converted into the sequence with some predetermined pulse width t_W and sharp edges. If the pulse width t_W is taken long enough, the application can reliably process an input signal.

Such conversion may be implemented using a simple circuit known as “monostable multivibrator”. One possible solution using a versatile chip 74HC4538 is shown in **Fig.16**.

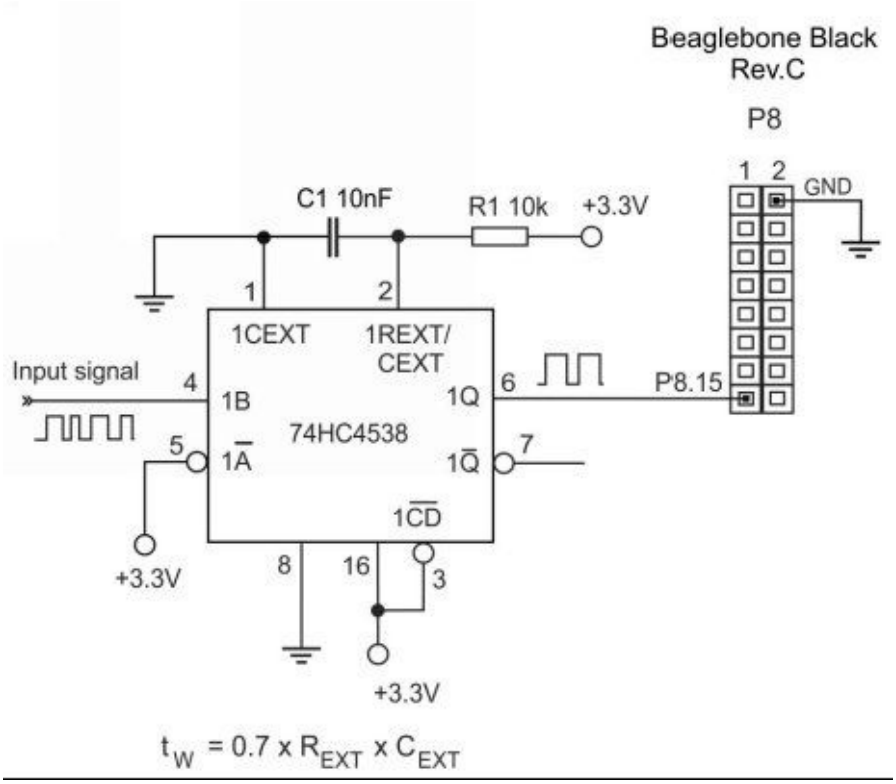


Fig.16

Let’s look at how this works. The 74HC4538 device is dual retriggerable-resettable monostable multivibrator. Each multivibrator has an active LOW trigger/retrigger input (**nA**), an active HIGH trigger/retrigger input (**nB**), an overriding active LOW direct reset input (**nCD**), an output (**nQ**) and its complement, and two pins (**nREXT/CEXT** and **nCEXT**) for connecting the external timing components $\mathbf{C_{EXT}}$ and $\mathbf{R_{EXT}}$. Typical pulse width variation over the specified temperature range is ±0.2 %.

The multivibrator may be triggered by either the positive or the negative edges of the input pulse. The duration and accuracy of the output pulse are determined by the external timing components $\mathbf{C_{EXT}}$ and $\mathbf{R_{EXT}}$.

The output pulse width $\mathbf{t_W}$ is equal to

$0.7 \times R_{EXT} \times C_{EXT}$

Given $\mathbf{R_{EXT}}$ and $\mathbf{C_{EXT}}$, the pulse width $\mathbf{t_W}$ will be 70 microseconds (70 µS).

The linear design techniques guarantee precise control of the output pulse width. A LOW level on **nCD** terminates the output pulse immediately. Schmitt trigger action on pins **nA** and **nB** makes the circuit highly tolerant of slower rise and fall times.

Alternatively, we can apply the 74HC123 chip. The connections of the circuit with this chip are shown below (**Fig.17**).

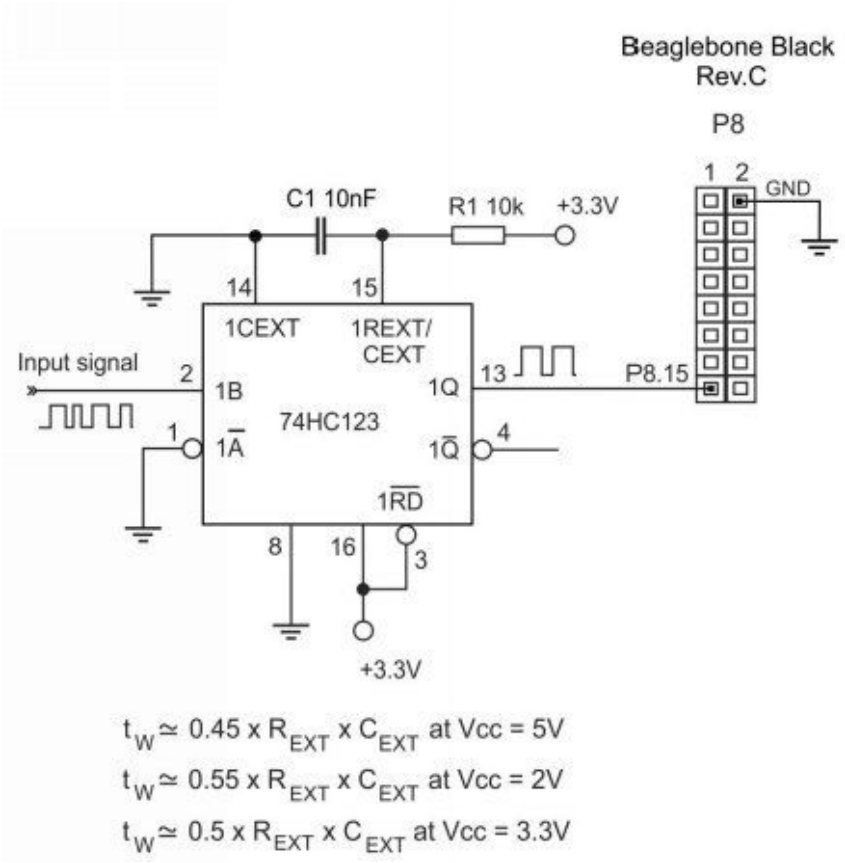


Fig.17

In this circuit, the pulse width $\mathbf{t_W}$ of the output signal depends also on the supply voltage and in the case of 3.3V will be

$$t_W = 0.5 \times R_{EXT} \times C_{EXT}$$

Often the input signal is far from the pure digital waveform and/or has small amplitude which makes impossible to process it by digital circuits. In those cases, the simple circuit arranged around an analog comparator IC helps to solve the problem (**Fig.18**).

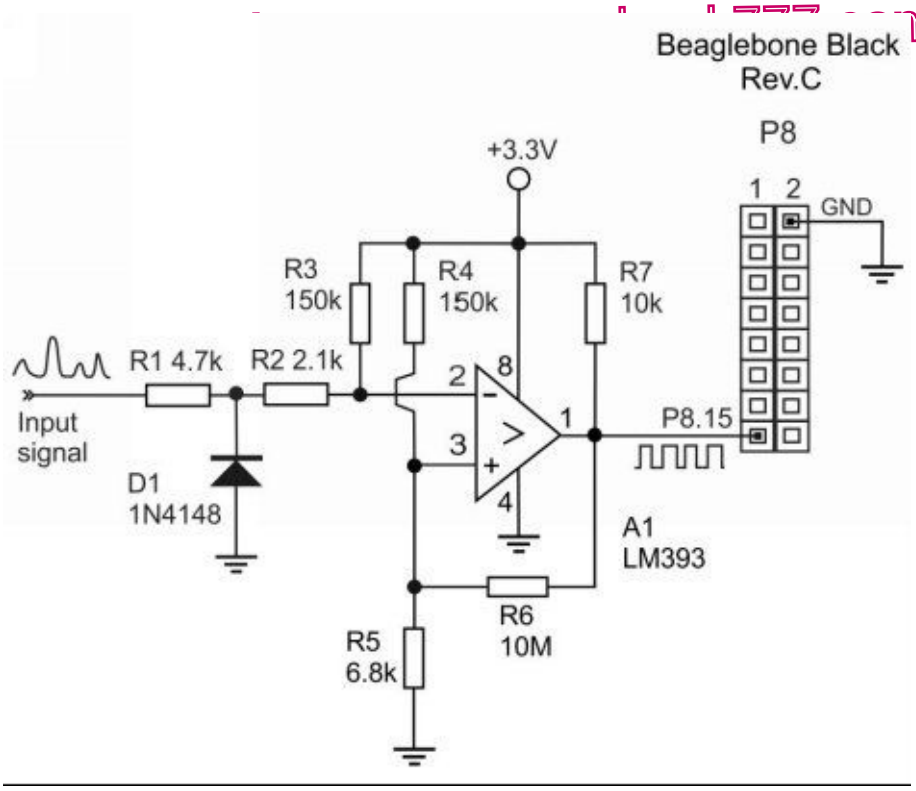


Fig.18

This circuit is arranged as a zero-crossing detector built around the comparator LM393. The threshold is determined by the resistive divider R4 – R5, the diode D1 clips the signal when it goes negative. The LM393 can be replaced by any general-purpose comparator (LMC7221, LMC7211, etc.).

Sometimes a developer needs to read a few digital inputs at a time and save a result in some variable. Assume we want to read 8 data bits from the port **P8**. The signals to be read (labeled D7 – D0) are shown in **Fig.19**.

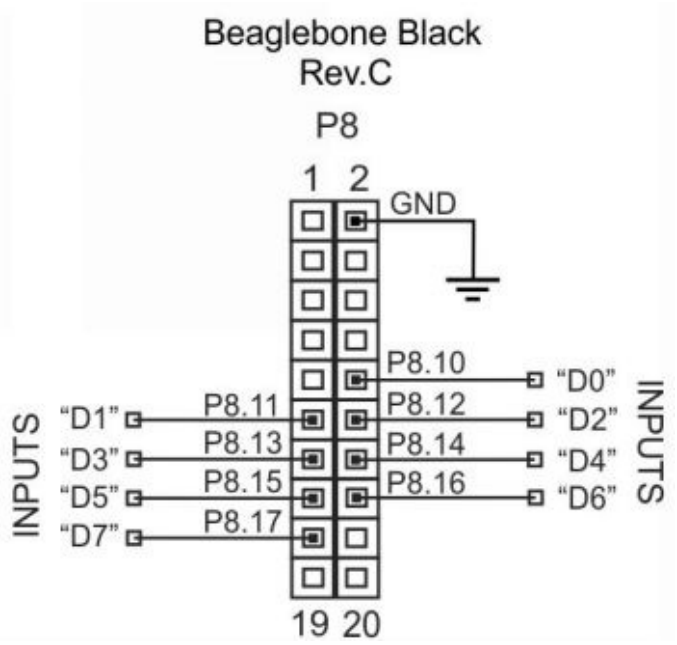


Fig.19

The following source code allows to read these inputs (**Listing 2**).

Listing 2.

```
import Adafruit_BBIO.GPIO as GPIO
import array
import time

QXX = [“P8_17”,“P8_16”,“P8_15”,“P8_14”,“P8_13”,“P8_12”,“P8_11”,“P8_10”]
L1 = [0,0,0,0,0,0,0,0]

tmp = 0

GPIO.setup(“P8_10”, GPIO.IN) # data bit 0
GPIO.setup(“P8_11”, GPIO.IN) # data bit 1
GPIO.setup(“P8_12”, GPIO.IN) # data bit 2
GPIO.setup(“P8_13”, GPIO.IN) # data bit 3
GPIO.setup(“P8_14”, GPIO.IN) # data bit 4
GPIO.setup(“P8_15”, GPIO.IN) # data bit 5
```

```
GPIO.setup(“P8_16”, GPIO.IN) # data bit 6
GPIO.setup(“P8_17”, GPIO.IN) # data bit 7

tmp = 0
for i in range(len(QXX)):
    L1[i] = GPIO.input(QXX[i])
for i in range(len(L1)):
    tmp |= L1[i] << len(L1)-1-i
print “Bits D7-D0 on port P8: ” + bin(tmp)
```


In this code, the elements of the array QXX are GPIO pins configured as inputs. The state of each input is saved in the element of the array L1 with the corresponding index by the following for() loop:

```
for i in range(len(QXX)):
    L1[i] = GPIO.input(QXX[i])
```

After all inputs have been processed, the 8-bit result will be put in the tmp variable after executing the following for() loop:

```
for i in range(len(L1)):
    tmp |= L1[i] << len(L1)-1-i
```

The running application produces the following result:

```
root@beaglebone:/# python Read_8bit.py
Bits D7-D0 on port P8: 0b11011100
root@beaglebone:/# python Read_8bit.py
Bits D7-D0 on port P8: 0b10011000
root@beaglebone:/# python Read_8bit.py
Bits D7-D0 on port P8: 0b10111000
root@beaglebone:/# python Read_8bit.py
```

Bits D7-D0 on port P8: 0b10011000

The performance of an application that mixes input, calculation, and output may be improved by running these as three separate threads. When we need to execute some tasks in parallel, we can read digital inputs in the separate thread as is illustrated by the simple source code in **Listing 3**.

Listing 3.

```
import Adafruit_BBIO.GPIO as GPIO
import threading
import array
import time

QXX = [“P8_17”,“P8_16”,“P8_15”,“P8_14”,“P8_13”,“P8_12”,“P8_11”,“P8_10”]
L1 = [0,0,0,0,0,0,0,0]

tmp = 0
Done = 0

GPIO.setup(“P8_10”, GPIO.IN) # D0
GPIO.setup(“P8_11”, GPIO.IN) # D1
GPIO.setup(“P8_12”, GPIO.IN) # D2
GPIO.setup(“P8_13”, GPIO.IN) # D3
GPIO.setup(“P8_14”, GPIO.IN) # D4
GPIO.setup(“P8_15”, GPIO.IN) # D5
GPIO.setup(“P8_16”, GPIO.IN) # D6
GPIO.setup(“P8_17”, GPIO.IN) # D7

class InpDataThread(threading.Thread):
    def run(self):
    global tmp
```

```
    global Done
    cnt = 0
    while cnt < 10:
    tmp = 0
    for i in range(len(QXX)):
    L1[i] = GPIO.input(QXX[i])
    for i in range(len(L1)):
    tmp |= L1[i] << len(L1)-1-i
    print str(cnt+1) +": Bits D7-D0 on port P8: " + bin(tmp)
    cnt = cnt + 1
    time.sleep(5)
    Done = 1


thRead = InpDataThread()
thRead.start()
while Done == 0:
    <some task is executed here>
    continue
thRead.join()
```


While the basic thread of an application performs some useful job, the **InpDataThread** thread periodically reads the data from the **P8** pins and outputs it to the console. When the counter (the **cnt** variable) reaches 10, the **InpDataThread** thread terminates.

The running application provides the following output:

```
root@beaglebone:/# python THRead_8bit.py
1: Bits D7-D0 on port P8: 0b11011001
2: Bits D7-D0 on port P8: 0b11011001
3: Bits D7-D0 on port P8: 0b11111001
4: Bits D7-D0 on port P8: 0b11111001
5: Bits D7-D0 on port P8: 0b11110001
6: Bits D7-D0 on port P8: 0b11110001
```

7: Bits D7-D0 on port P8: 0b11100001

8: Bits D7-D0 on port P8: 0b11100001

9: Bits D7-D0 on port P8: 0b11100001

10: Bits D7-D0 on port P8: 0b11100001

Most BeagleBone applications need to operate with timing intervals in some way. Usually, developers apply the **sleep()** function from the time module which allows to implement delays within a program.

Often, the **sleep()** doesn't fit your needs, especially when you want to operate with precision and/or short intervals. This is because the **sleep()** function is realized in software and highly depends on the performance of an operating system. The accuracy of time intervals provided by the **sleep()** function can vary depending on the many factors including running processes and threads.

When we need to approach our application to real time, we can use, for example, some external clock source for synchronizing time intervals. This clock source should be independent of the operating system and can be implemented in hardware.

The next example illustrates this concept. Here the 8-bit data is periodically read from **P8**. When the state of any of input pins changes as compared to its previous state, this is detected by the program. The interval between adjacent read operations is determined by an external clock source (an oscillator circuit). The rising edge of each clock pulse arriving on pin **P8.18** asynchronously starts a separate task.

The circuit diagram of this project is shown in **Fig.20**.

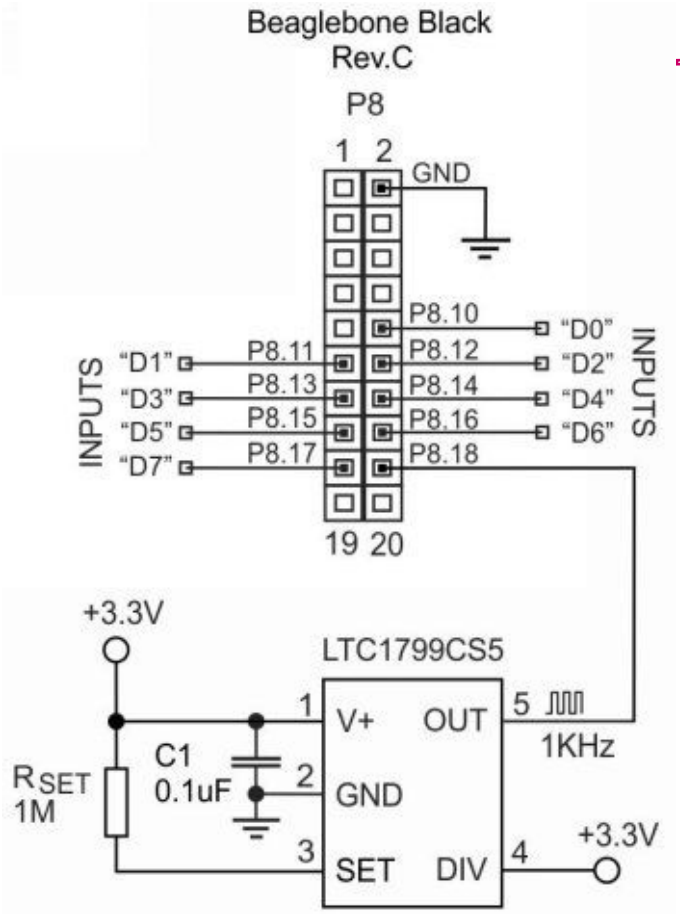


Fig.20

In this circuit, the oscillator is built around the LTC1799 chip which provides the high precision pulse train of 1KHz on its output (pin 5). The period of these clock pulses is, therefore, 1 mS. The separate thread which reads 8-bit data from the port **P8** is created every 1 mS.

Note that the time interval taken for processing some task (1mS, in our case) should be long enough, so that the program code within a thread would terminate before a next clock pulse arrives. As a clock source, you can employ any of numerous circuits (for example, based upon using a timer 555).

The Python source code for this project is shown in **Listing 4**.

Listing 4.

```
import Adafruit_BBIO.GPIO as GPIO
import threading
import time
import math
```

```
QXX = [“P8_17”,“P8_16”,“P8_15”,“P8_14”,“P8_13”,“P8_12”,“P8_11”,“P8_10”]
L1 = [0,0,0,0,0,0,0,0]

ioState = 0
tmp = 0
changed = 0

GPIO.setup(“P8_10”, GPIO.IN) # D0
GPIO.setup(“P8_11”, GPIO.IN) # D1
GPIO.setup(“P8_12”, GPIO.IN) # D2
GPIO.setup(“P8_13”, GPIO.IN) # D3
GPIO.setup(“P8_14”, GPIO.IN) # D4
GPIO.setup(“P8_15”, GPIO.IN) # D5
GPIO.setup(“P8_16”, GPIO.IN) # D6
GPIO.setup(“P8_17”, GPIO.IN) # D7

GPIO.setup(“P8_18”, GPIO.IN) # 1mS period pulse train

GPIO.add_event_detect(“P8_18”, GPIO.RISING)

class EvThread(threading.Thread):
    def run(self):
    global ioState
    global tmp
    global changed
    ioState = 0
    for i in range(len(QXX)):
    L1[i] = GPIO.input(QXX[i])
    for i in range(len(L1)):
    ioState |= L1[i] << len(L1)-1-i
    if ioState != tmp:
```

```
    tmp = ioState
    changed = 1
```


```
for i in range(len(QXX)):
    L1[i] = GPIO.input(QXX[i])
for i in range(len(L1)):
    ioState |= L1[i] << len(L1)-1-i
tmp = ioState
print “Bits D7-D0 on port P8: ” + bin(tmp)


while True:
    if GPIO.event_detected(“P8_18”):
    a = EvThread()
    a.start()
    a.join()
    if changed == 1:
    print “Bits D7-D0 on port P8: ” + bin(tmp)
    changed = 0
```

The next topic to discuss is about reading signals from external circuitry. When dealing with high-power and/or noisy circuits, we can apply optoisolator circuits for reading the digital input signals. We have already discussed those circuits, so let’s look at how to interface optoisolators with sensors.

Despite the fact that most optoisolator devices are rated for operating at +5V, they can operate at +3.3V. This essentially simplifies interfacing optoisolators to the BeagleBone Black inputs.

The next circuit diagrams (**Fig.21** – **Fig.22**) illustrate possible ways to connect digital sensors to the BeagleBone Black using optoisolators.

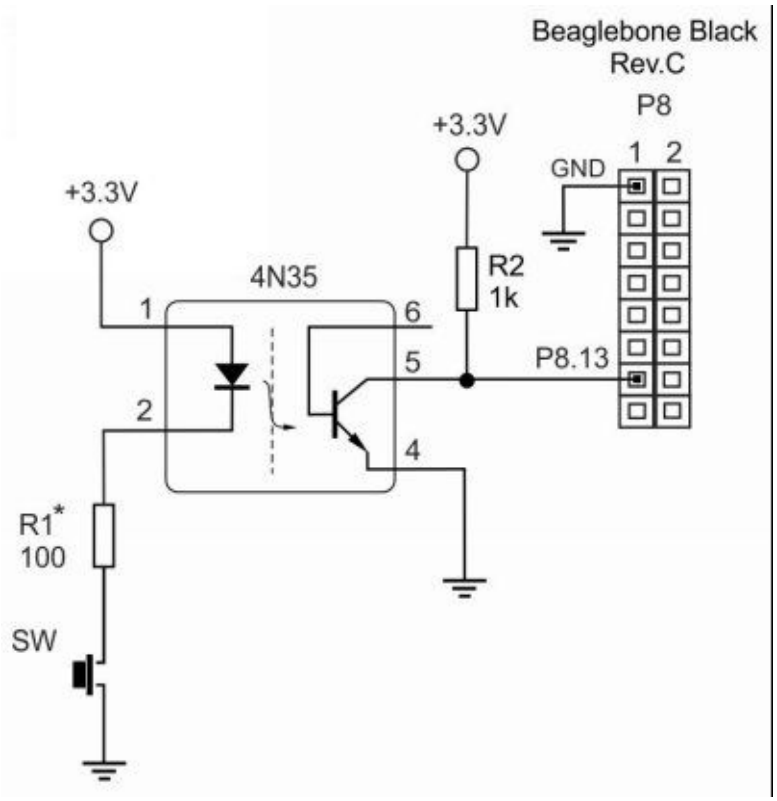


Fig.21

In this circuit, the digital sensor is simulated by the mechanical switch **SW**. When **SW** is pressed, the LED is ON and the signal on pin 5 of 4N35 optoisolator goes low. When **SW** is released, pin 5 goes high. The state of pin 5 of 4N35 optoisolator can be read on pin **P8.13** of the BeagleBone Black.

Instead of the 4N35 device we can also apply the high-speed optoisolator 6N137 or HCPL2601 as is shown in **Fig.22**.

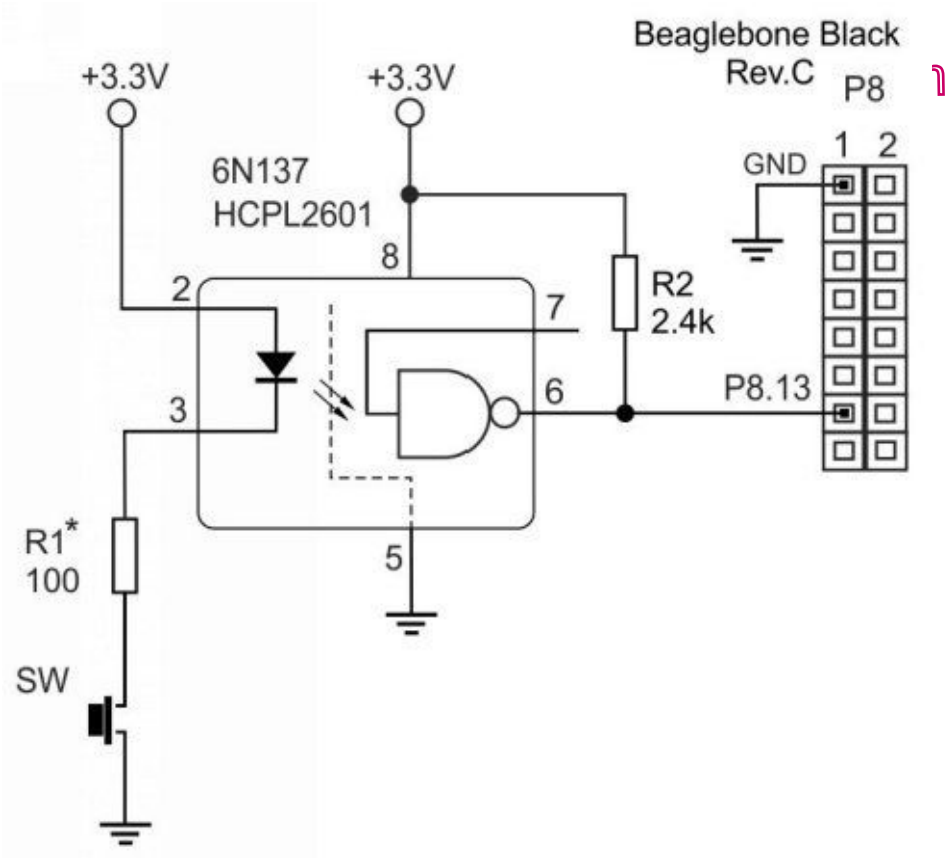

Beaglebone Black
Rev.C
P8
+3.3V
+3.3V
6N137
HCPL2601
GND
1 2
8
2
7
R2
2.4k
3
6
P8.13
R1*
100
5
SW

Fig.22

BeagleBone Black Networking: project 1

Since the BeagleBone Black is driven by Debian OS, it is possible to use the BeagleBone as an “intellectual” node which will transfer/receive data being measured to/from other computers over a TCP/IP network. In this communication, at least two network applications will be involved.

One of these should be a “TCP client” which send data to or receive data from the other application running as a “TCP server”. Usually, the TCP client and TCP server are running on different physical computers, although both can be launched on the same host using a “loopback” interface with an IP address 127.0.0.1.

Communications between TCP/IP network applications are implemented using *sockets*. A *socket* is a communication mechanism that allows client/server systems to be developed either locally, on a single machine, or across networks. Sockets are the endpoints of a bidirectional communication channel. Sockets may communicate within the same process, between processes on the same machine, or between processes on different computers .Many Linux functions such as printing, connecting to databases, and serving web pages as well as network utilities such as rlogin for remote login and ftp for file transfer usually use sockets to communicate. The socket mechanism can implement multiple clients attached to a single server.

In this section, we will discuss two examples. The first will illustrate the case when data are sent from the TCP client to the TCP server. The second example will show how the TCP client can receive data from the TCP server. Both TCP server and TCP client will be running on the same PC (the BeagleBone Black) using the “loopback” interface with an IP address 127.0.0.1.

The data will be taken from pins **P8.10** – **P8.17** of the BeagleBone Black (the source code for this case has been discussed earlier).

The client side of a socket-based system is very simple and straightforward. The client creates a socket by calling the **socket()** function. This is followed by the **connect()** function to establish a connection with the TCP server by using the server’s listening socket. Once established, sockets can be used like low-level file descriptors, providing two-way data communications. The data can be transferred from the TCP client to the TCP server using the **send()** function. To receive data from the server the **recv()** function can be used.

The next source code (**Listing 5**) represents the simple TCP client application which will transfer data to the TCP server. For the sake of brevity, error checking is omitted.

However, in production code you should always check for error returns unless there is a very good reason to omit this check.

Listing 5.

```
import Adafruit_BBIO.GPIO as GPIO
import socket
import threading
import array
import time

QXX = [“P8_17”,“P8_16”,“P8_15”,“P8_14”,“P8_13”,“P8_12”,“P8_11”,“P8_10”]
L1 = [0,0,0,0,0,0,0,0]

tmp = 0
port = 7777
Done = 0

GPIO.setup(“P8_10”, GPIO.IN) # D0
GPIO.setup(“P8_11”, GPIO.IN) # D1
GPIO.setup(“P8_12”, GPIO.IN) # D2
GPIO.setup(“P8_13”, GPIO.IN) # D3
GPIO.setup(“P8_14”, GPIO.IN) # D4
GPIO.setup(“P8_15”, GPIO.IN) # D5
GPIO.setup(“P8_16”, GPIO.IN) # D6
GPIO.setup(“P8_17”, GPIO.IN) # D7

class InpDataThread(threading.Thread):
    def run(self):
    global tmp
    global port
    global Done
```

```
    cnt = 0
    while cnt < 10:
    tmp = 0
    for i in range(len(QXX)):
    L1[i] = GPIO.input(QXX[i])
    for i in range(len(L1)):
    tmp |= L1[i] << len(L1)-1-i
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = socket.gethostname()
    s.connect((host, port))
    s.send(str(cnt+1) +": Bits D7-D0 on port P8: " + bin(tmp))
    s.close()
    cnt = cnt + 1
    time.sleep(5)
    Done = 1

thRead = InpDataThread()
thRead.start()
while Done == 0:
    continue
thRead.join()
```

This code is mostly known for us except the section responsible for the network communication. This is shown below:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
s.connect((host, port))
s.send(str(cnt+1) +": Bits D7-D0 on port P8: " + bin(tmp))
s.close()
```

First, we create a socket (variable **s**) using the **socket.socket()** function available in the

socket module. Then the program obtains the local machine name of the computer where the TCP client should be connected to. This is done by the

host = socket.gethostname()

function.

To connect to the TCP server we should call the **connect()**, passing address **host** and **port** as the parameters to this function:

s.connect((host, port))

Each server or client should use one or more ports. A port may be identified by a number, a string containing a number, or the name of a service.

The **send()** function that follows transfers the message to the TCP server. Finally, the socket is closed by invoking the **close()** function.

Let’s look at how to develop a TCP server application.

First, the server application creates a socket, using the **socket()** function the **socket** module. Next, the server process gives the socket a name (**host** and **port**) by invoking the **bind()** function.

The server application then waits for a client to connect to the named socket. The **listen()** function creates a queue for incoming connections. The server can accept them using the **accept()** function.

When the server calls **accept()**, a new socket is created that is distinct from the named socket. This new socket is used solely for communication with this particular client. The named socket remains for further connections from other clients. If the server code is written appropriately, it can take advantage of multiple connections.

For the simple TCP server, further clients wait on the listen queue until the server is ready again.

The TCP server source code running on the same BeagleBone host is shown in **Listing 6**.

Listing 6.

```
import socket               # import the socket module
```

```
srv = socket.socket()              # Create a socket object
host = socket.gethostname()   # Get local machine name
port = 7777                          # Assign a port to receive incoming requests
addr = (host, port)
srv.bind(addr)
srv.listen(5)
print “TCP Server is waiting for incoming requests on port 7777…”

while True:
    c, caddr = srv.accept()
    msg = c.recv(256)
    print msg
    c.close()
```

The program code creates the socket by executing the statement

```
srv = socket.socket()
```

The **socket()** function returns a descriptor in the variable **srv** – it can be used for accessing the socket. Then the local machine name is obtained by invoking the

```
host = socket.gethostname()
```

function. The value returned by this function is saved in the **host** variable. The TCP server also needs the port (we use 7777). Both the host and port are needed for making the socket available for use by application. The **addr** variable makes up the address needed for the socket from the **host** and **port**:

```
addr = (host, port)
```

Then the program code invokes the **bind()** function which assigns the address specified in the **addr** variable to the unnamed socket associated with the file descriptor **srv**.

To accept incoming connections on a socket, a server program must create a queue to store pending requests. It accomplishes this by using the **listen()** function.

After creating the listening socket, the server enters the endless **while()** loop waiting for incoming requests on port 7777. When a new request comes, the TCP server creates a working connection (socket) to the particular client by invoking the **accept()** function:

```
c, caddr = srv.accept()
```

The **accept()** function returns the client connection ID (variable **c**) and the address assigned to a working socket (variable **caddr**). When done, all data are passed through this connection. In our case, the TCP server receives data from the client by invoking the **recv()** function:

```
msg = c.recv(256)
```

Then data obtained (variable **msg**) is output to the console, the working socket is closed by invoking the (**close()** function) and the **while()** loop continues.

The TCP server provides the following output:

```
root@beaglebone:/# python TCPServer_Read8bit.py
TCP Server is waiting for incoming requests on port 7777…
1: Bits D7-D0 on port P8: 0b11101001
2: Bits D7-D0 on port P8: 0b11101001
3: Bits D7-D0 on port P8: 0b11101001
4: Bits D7-D0 on port P8: 0b11101001
5: Bits D7-D0 on port P8: 0b11001001
6: Bits D7-D0 on port P8: 0b11101001
7: Bits D7-D0 on port P8: 0b11001000
8: Bits D7-D0 on port P8: 0b11001000
9: Bits D7-D0 on port P8: 0b11001000
10: Bits D7-D0 on port P8: 0b11001000
```

BeagleBone Black Networking: project 2

Another client-server configuration is presented in this section. Here the TCP server waits for incoming requests from the TCP client and when the connection is established sends data to the client.

Both the client and server source code are much the same as in the project 1, except the data flow direction – in this configuration the TCP server sends data to the TCP client.

In this configuration, the TCP server and client use the port 8888 to establish a communication.

The Python source code for the TCP server is shown in **Listing 7**.

Listing 7.

```
import Adafruit_BBIO.GPIO as GPIO
import socket              # import the socket module
import array

QXX = [“P8_17”,“P8_16”,“P8_15”,“P8_14”,“P8_13”,“P8_12”,“P8_11”,“P8_10”]
L1 = [0,0,0,0,0,0,0,0]

port = 8888

GPIO.setup(“P8_10”, GPIO.IN) # D0
GPIO.setup(“P8_11”, GPIO.IN) # D1
GPIO.setup(“P8_12”, GPIO.IN) # D2
GPIO.setup(“P8_13”, GPIO.IN) # D3
GPIO.setup(“P8_14”, GPIO.IN) # D4
GPIO.setup(“P8_15”, GPIO.IN) # D5
GPIO.setup(“P8_16”, GPIO.IN) # D6
GPIO.setup(“P8_17”, GPIO.IN) # D7

srv = socket.socket()        # Create a socket object
```

```
host = socket.gethostname()   # Get a local machine name

addr = (host, port)
srv.bind(addr)
srv.listen(5)
print “TCP Server is waiting for incoming requests on port 8888…”

while True:
    tmp = 0
    cl, caddr = srv.accept()
    for i in range(len(QXX)):
    L1[i] = GPIO.input(QXX[i])
    for i in range(len(L1)):
    tmp |= L1[i] << len(L1)-1-i
    cl.send(“Bits D7-D0 on port P8: ” + bin(tmp))
    cl.close()
```

The TCP-client is presented by the following source code (**Listing 8**).

Listing 8.

```
import socket

port = 8888
cl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
cl.connect((host, port))
msg = cl.recv(256)
print msg
cl.close()
```

Distributed systems using Tiva C Series Connected LaunchPad: project1

This section is dedicated to the design of the simple distributed measurement system comprising the BeagleBone Black and the Tiva™ C Series Connected LaunchPad evaluation board from Texas Instruments. The Tiva™ C Series Connected LaunchPad can be used as an “intellectual” node processing signals from sensors located somewhere on the network.

Below there is an excerpt taken from the **User’s Guide on Tiva™ C Series Connected LaunchPad**.

The Tiva™ C Series TM4C1294 Connected LaunchPad Evaluation Board (EK-TM4C1294XL) is a low-cost evaluation platform for ARM® Cortex™-M4F-based microcontrollers. The Connected LaunchPad design highlights the TM4C1294NCPDT microcontroller with its on-chip 10/100 Ethernet MAC and PHY, USB 2.0,hibernation module, motion control pulse-width modulation and a multitude of simultaneous serial connectivity.

The Connected LaunchPad also features two user switches, four user LEDs, dedicated reset and wake switches, a breadboard expansion option and two independent BoosterPack XL expansion connectors. The pre-programmed quick start application on the Connected LaunchPad also enables remote monitoring and control of the evaluation board from an internet browser anywhere in the world. The board allows users to create and customize their own Internet-of-Things applications.

Fig.23 shows a photo of the Connected LaunchPad with key features highlighted.

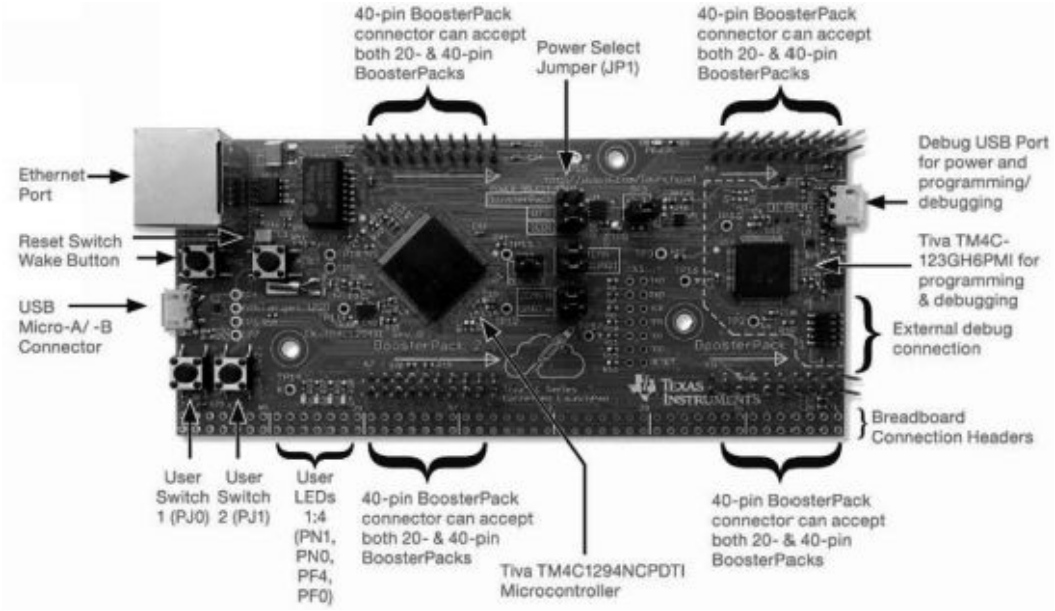


Fig.23

The Connected LaunchPad is designed to connect directly to an Ethernet network using RJ45 style connectors. The microcontroller contains a fully integrated Ethernet MAC and PHY. This integration creates a simple, elegant and cost-saving Ethernet circuit design. Example code is available for both the uIP and LwIP TCP/IP protocol stacks. The embedded Ethernet on this device can be programmed to act as an HTTP server, client or both. The design and integration of the circuit and microcontroller also enable users to synchronize events over the network using the IEEE1588 precision time protocol.

When configured for Ethernet operation, it is recommended that the user configure LED D3 and D4 to be controlled by the Ethernet MAC to indicate connection and transmit/receive status. More information about LaunchPad can be found on the TI site www.ti.com.

The following project will illustrate how to read data from an analog sensor attached to the channel **A0** of the Connected LaunchPad and then observe the result using the WEB-browser running on the BeagleBone Black. The diagram of the distributed measurement system consisting of the BeagleBoneBlack and the Connected LaunchPad is shown in **Fig.24**.

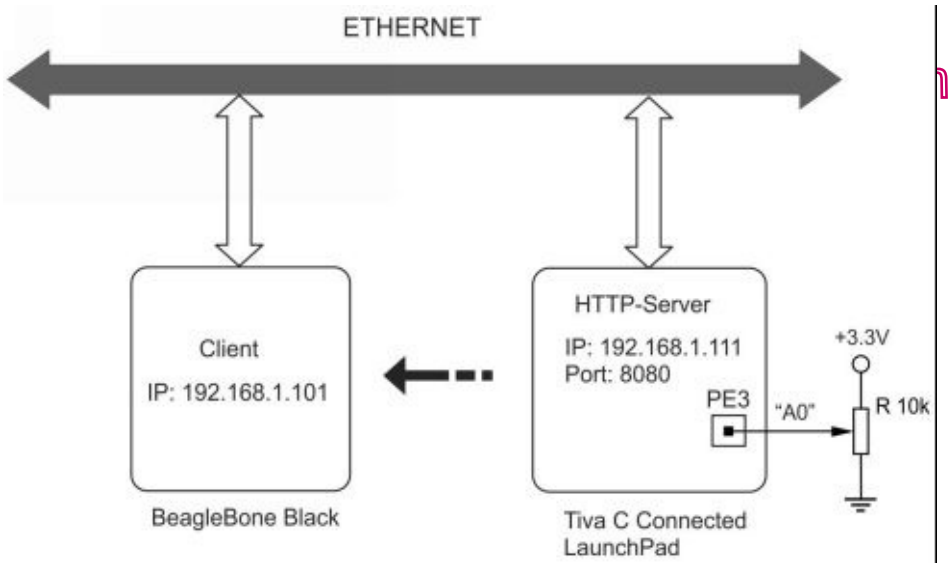


Fig.24

In this configuration, the simple HTTP (Web) Server is running on the Connected LaunchPad. The IP-address of the server is 192.168.1.111; the server is listening for incoming requests on port 8080.

When the request comes from the client running on the BeagleBoneBlack, the HTTP server running on the Connected LaunchPad reads the voltage on the analog input **A0** (pin **PE3**), then passes the result back to the TCP client. In our case, the analog input voltage on A0 is simulated by the voltage divider formed by the potentiometer R.

The BeagleBone client application can be a Web-browser through which we can observe the data obtained by the HTTP server. The browser window can look like the following (**Fig.25**).

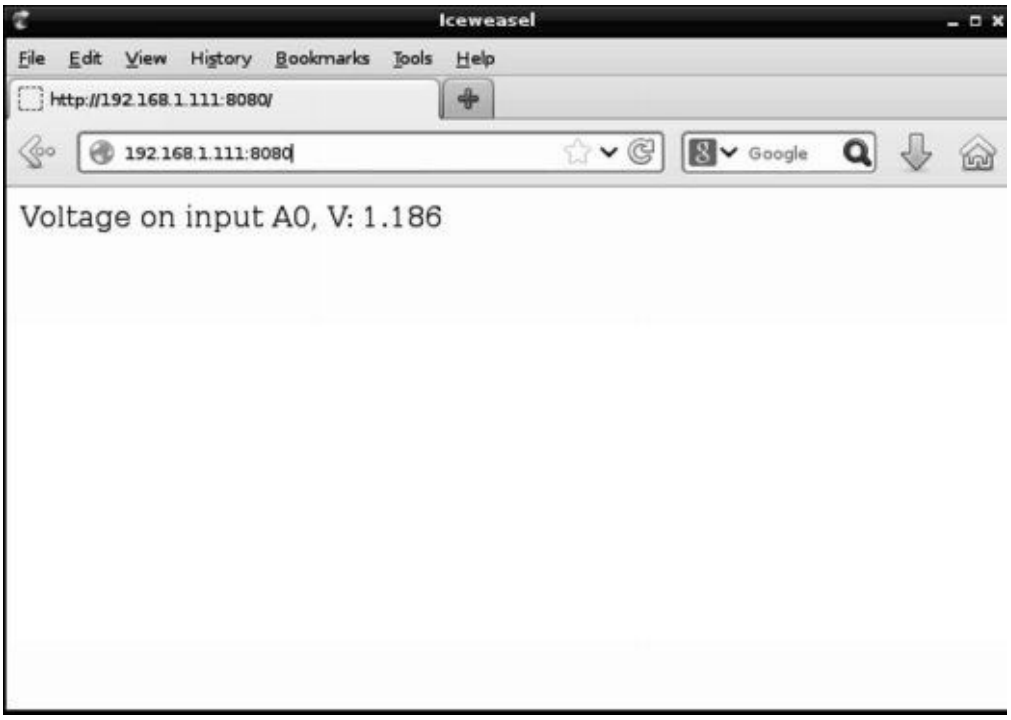


Fig.25

For developing the Connected LaunchPad applications the open-source prototyping platform called Energia was used. The Energia uses the Wiring and Arduino framework for programming Texas Instruments launchpads. The Energia IDE is the cross platform and supported on Mac OS, Windows, and Linux. Energia includes an integrated development environment (IDE) that is based on Processing.

Together with Energia, LaunchPad can be used to develop interactive objects, taking inputs from a variety of switches or sensors, and controlling a variety of lights, motors, and other physical outputs. LaunchPad projects can be stand-alone (only run on the Target Board, i.e. your LaunchPad), or they can communicate with software running on your computer (Host PC). Much more details about Energian can be found on http://energia.nu.

After launching, the Energia application window will look like the following (**Fig.26**). For developing the Connected LaunchPad application the version 0101E0013 of Energia was used.

Fig.26

The Connected LaunchPad source code for the HTTP-server is shown in **Listing 9**.

Listing 9.

```
#include <Ethernet.h>

float inpVal = 0;  // variable to store the value coming from the sensor
int bCode = 0;
const float VREF = 3.36;

// assign a MAC-address for the Ethernet controller.
// fill in the MAC-address of your Connected LaunchPad board here:
byte mac[] = {0x00, 0x1A, 0xB6, 0x02, 0xAE, 0xCD};
// assign an IP address for the controller:
byte ip[] = {192,168,1,111};
byte gateway[] = {192,168,1,1};
```

```
byte subnet[] = {255, 255, 255, 0};



// Initialize the Ethernet server library
// with the IP address and port you want to use
// we take port 8080 for our HTTP-server:
EthernetServer server(8080);



void setup() {
    // start the Ethernet connection and the server:

  Ethernet.begin(mac, ip);
  server.begin();
  Serial.begin(9600);

  // give the Ethernet interface some time to set up:
  delay(1000);
}

void loop() {
    // listening for incoming Ethernet connections
  listenForEthernetClients();
}

void listenForEthernetClients() {
  // listen for incoming requests
  EthernetClient client = server.available();
  if (client) {
    Serial.println("A new client connection is established.");
    // an HTTP-request ends with a blank line
```

```
boolean currentLineIsBlank = true;
while (client.connected()) {
if (client.available()) {
char c = client.read();
// if you've gotten to the end of the line (received a newline
// character) and the line is blank, the http request has ended,
// so you can send a reply
if (c == '\n' && currentLineIsBlank) {
// send a standard HTTP response header
client.println("HTTP/1.1 200 OK");
client.println("Content-Type: text/html");
client.println();
// print the current readings in HTML format:
bCode = analogRead(A0);
inpVal = VREF/4096*bCode;
client.print("Voltage Level on input A0: ");
client.print(inpVal, 3);
client.print(" V");
client.println("<br />");
break;
}
if (c == '\n') {
// you're starting a new line
currentLineIsBlank = true;
}
else if (c != '\r') {
// you've gotten a character on the current line
currentLineIsBlank = false;
}
}
}
```

```
    // give the web browser time to receive the data
    delay(1);
    // close the working connection:
    client.stop();
  }
}
```

In this particular application, we can select the static IP-address 192.168.1.111 and port 8080 for our HTTP-server. You can select the proper IP-address and port for your particular application, of course.

To write the code we used the Ethernet library, similar to that applied in Arduino network applications. The library supports up to four concurrent connection (incoming or outgoing or a combination). The above source code can be easily modified for performing other task such as reading digital signals on GPIO pins of the Connected LaunchPad.

When compiling the source code, you can receive the following error messages:

C:\ENERGIA\hardware\lm4f\libraries\Ethernet/Ethernet.h:12:28: error: redefinition of ‘const IPAddress INADDR_NONE’

. . .

C:\ENERGIA\hardware\lm4f\cores\lm4f/IPAddress.h:73:17: error: ‘const IPAddress INADDR_NONE’ previously declared here

I assume this may be the bug, so you need to edit the Ethernet.h file by commenting out the following line using /* */ :

```
/* const IPAddress INADDR_NONE(0,0,0,0); */
```

When done, you can recompile the source code – the error will not appear.

To be aware that your server is running properly, you can simply enter the **ping** command from either host on the local network. As the HTTP (TCP) server is OK, you will observe the response like this:

```
root@beaglebone:/# ping 192.168.1.111
PING 192.168.1.111 (192.168.1.111) 56(84) bytes of data.
```

64 bytes from 192.168.1.111: icmp_req=1 ttl=255 time=0.543 ms

64 bytes from 192.168.1.111: icmp_req=2 ttl=255 time=0.264 ms

64 bytes from 192.168.1.111: icmp_req=3 ttl=255 time=0.280 ms

64 bytes from 192.168.1.111: icmp_req=4 ttl=255 time=0.279 ms

64 bytes from 192.168.1.111: icmp_req=5 ttl=255 time=0.275 ms

Distributed systems using Tiva C Series Connected LaunchPad: project 2

This section describes the network system where the GPIO pins of the Connected LaunchPad are controlled by the Python network application running on the BeagleBone Black. The block diagram of this system is shown in **Fig.27**.

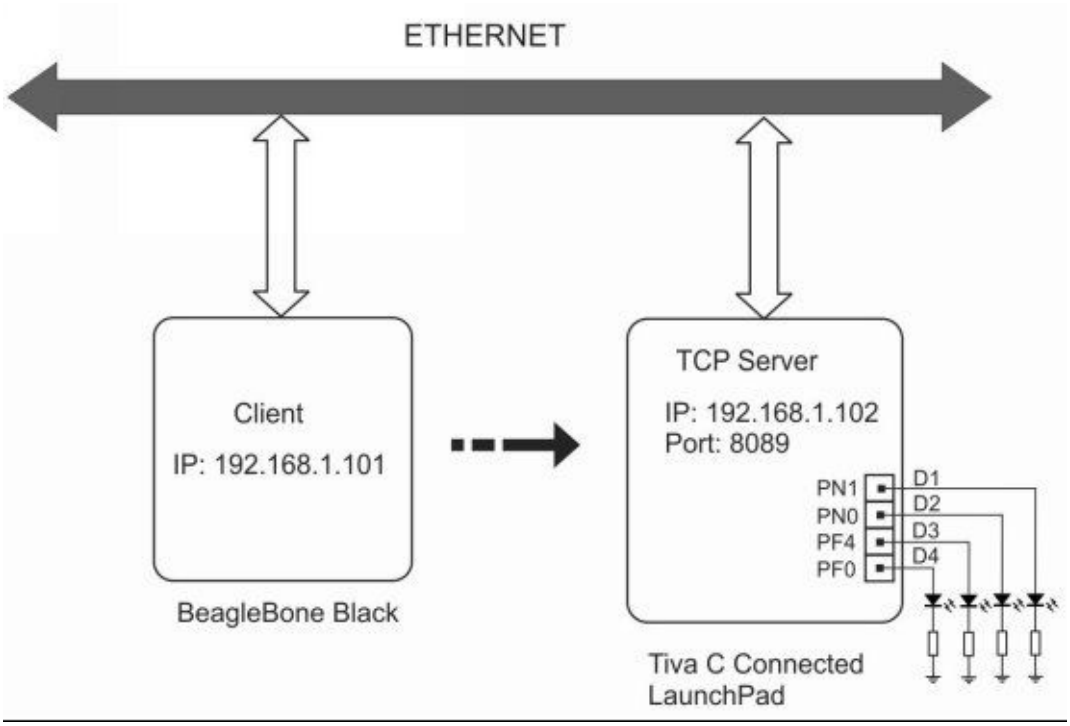


Fig.27

In this system, the TCP server running on the Tiva C Series Connected LaunchPad waits until the command comes from the TCP client running on the BeagleBone Black. The command itself is a number between 0 and 4. The command “0” drives the Connected LaunchPad on-board LEDS (D1 – D4) OFF; the command “1” drives D1 ON leaving the rest OFF, the command “2” drives D2 ON, etc.

The TCP server source code is shown in **Listing 10**.

Listing 10.

```
// TCP Server driving LEDS D1-D4

#include <Ethernet.h>
```

```
const int LED1 = PN_1;
const int LED2 = PN_0;
const int LED3 = PF_4;
const int LED4 = PF_0;

// assign a MAC address for the ethernet controller.
// fill in your address here:

byte mac[] = {0x00, 0x1A, 0xB6, 0x02, 0xAE, 0xCD};

// assign an IP address for the Ethernet controller:

IPAddress ip(192,168,1,111);              // the static IP: 192.168.1.111
IPAddress gateway(192,168,1,1);      // gateway: 192.168.1.1
IPAddress subnet(255, 255, 255, 0);  // subnet mask: 255.255.255.0



// Initializing the Ethernet server object

EthernetServer server(8089);

void setup() {
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);
  pinMode(LED4, OUTPUT);
  Serial.begin(9600);

  // start the Ethernet connection and the server:
```

```
  Serial.println("Trying to get an IP address using DHCP");
  if (Ethernet.begin(mac) == 0) {
    Serial.println("Failed to configure Ethernet using DHCP");
    // initialize the ethernet device using static IP-addresses
    Ethernet.begin(mac, ip, gateway, subnet);
  }
  // print the local IP-address of the Connected LaunchPad
  Serial.print("Connected LauncPad IP address: ");
  ip = Ethernet.localIP();
  for (byte ipByte = 0; ipByte < 4; ipByte++) {
    // print the value of each byte of the IP address:
    Serial.print(ip[ipByte], DEC);
    Serial.print(".");
  }
  Serial.println();

  server.begin();

  // give the Ethernet interface time to set up:
  delay(1000);
}

void loop() {
  // listen for incoming Ethernet connections:
  listenForEthernetClients();
}

void listenForEthernetClients() {
  // listen for incoming requests
  EthernetClient client = server.available();
  if (client) {
```

```
while (client.connected()) {
if (client.available()) {
char c = client.read();
int i1 = int(c - '0');
Serial.print(i1);
switch (i1) {
case 0:
digitalWrite(LED1, LOW);
digitalWrite(LED2, LOW);
digitalWrite(LED3, LOW);
digitalWrite(LED4, LOW);
break;
case 1:
digitalWrite(LED1, HIGH);
digitalWrite(LED2, LOW);
digitalWrite(LED3, LOW);
digitalWrite(LED4, LOW);
break;
case 2:
digitalWrite(LED1, LOW);
digitalWrite(LED2, HIGH);
digitalWrite(LED3, LOW);
digitalWrite(LED4, LOW);
break;
case 3:
digitalWrite(LED1, LOW);
digitalWrite(LED2, LOW);
digitalWrite(LED3, HIGH);
digitalWrite(LED4, LOW);
break;
case 4:
```

```
    digitalWrite(LED1, LOW);
    digitalWrite(LED2, LOW);
    digitalWrite(LED3, LOW);
    digitalWrite(LED4, HIGH);
    break;
    default:
    break;
    }
    }
    }
    delay(1);
    // close the connection:
    client.stop();
  }
}
```

In this application, the IP-address (192.168.1.102) is taken from the DHCP server. If it failed, the program would have used the predetermined static IP-address 192.168.1.111 instead. In our case, the DHCP server leased the IP-address 192.168.1.102 – this address is used by the TCP client to access the server. The listening socket is assigned port 8089.

The Python source code of the TCP client is shown in **Listing 11.**

Listing 11.

```
import socket
import time
cnt = 0
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
port = 8089
s.connect((“192.168.1.102”, port))
while cnt < 20:
```

```
    for led14 in range (0, 5):
    s.send(str(led14))
    time.sleep(0.25)
    cnt = cnt + 1
s.close()
```


This program connects to the TCP server using the IP-address 192.168.1.102 and port 8089. When connected, the program code sends the sequence of commands (0 through 4) via the TCP/IP network every 0.25 s. Each command affects the corresponding built-in LED on the Connected LaunchPad.

Measuring a frequency of digital signals

Measuring a frequency of digital signals is not a complicated task when you apply a microprocessor board capable of operating in real time. With BeagleBone Black such task is much more complicated because the board can't process signals in real-time unless the Programmable Realtime Units (PRUs) are used. Nevertheless, at low frequencies (up to a few KHz) it is possible to precisely measure this parameter.

The following project demonstrates it. The digital input signal to be measured will arrive on pin **P8.14** of the BeagleBone Black (**Fig.28**).

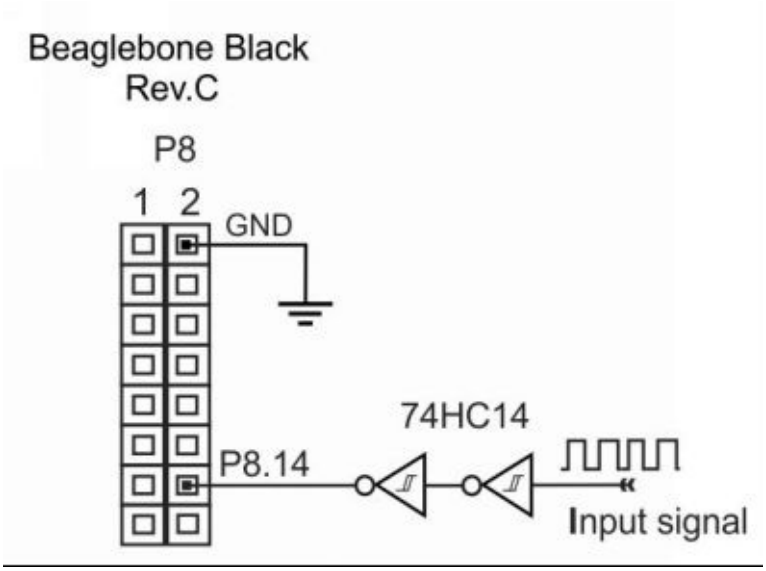


Fig.28

Here the buffer 74HC14 is optional – it is needed recover slow and/or distorted signals as they pass through long wires.

The Python source code for measuring the frequency of input signals is shown below (**Listing 12**).

Listing 12.

```
import Adafruit_BBIO.GPIO as GPIO
import threading
import time
import math
```

```
fCount = 0
Done = 0
total = 0


GPIO.setup("P8_14", GPIO.IN)

GPIO.add_event_detect("P8_14", GPIO.BOTH)

class CountThread(threading.Thread):
    def run(self):
    global fCount
    global Done
    fCount = 0
    while Done == 0:
    if GPIO.event_detected("P8_14"):
    fCount = fCount + 1

for cnt in range (0,19):
    Done = 0
    a = CountThread()
    a.start()
    time.sleep(1)
    Done = 1
    a.join()
    total = total + math.pow(fCount/2, 2)

total = math.sqrt(total/20)
total = round(total*1.035)
print "Input Frequency, Hz: " + str(total)
```

To measure the frequency of a pulse train the program code calculates the number of falling and rising edges passing through pin **P8.14** during 1 s. The single measurement iteration is performed by the following sequence:

```
Done = 0
a = CountThread()
a.start()
time.sleep(1)
Done = 1
a.join()
total = total + math.pow(fCount/2, 2)
```

Here the global variable **Done** is assigned 0 before the measurement cycle starts off. Then the separate thread **CountThread** is created and the pulse edges are counted until the 1 s interval has expired (the **sleep(1)** function). Then **Done** is assigned 1 thus terminating the **CountThread**.

To obtain the high-precision result we perform 20 measurement cycles. Then we get the final result through the following expressions:

```
total = math.sqrt(total/20)
total = round(total*1.035)
```

The coefficient 1.035 is empirical; it corrects the final result on the assumption that some edges have been missed because of time delays while executing the CPU instructions.

This system reliably measures the frequencies up to 3 – 4 KHz; at higher frequencies an error can be as large as 10 – 15 Hz.

Measuring high frequency signals

To measure frequencies up to a few hundred KHz, we can apply the digital counter/divider. The following project illustrates using a 74HC4040 IC that is a 12-stage binary ripple counter. The hardware circuit for this project is shown in **Fig.29**.

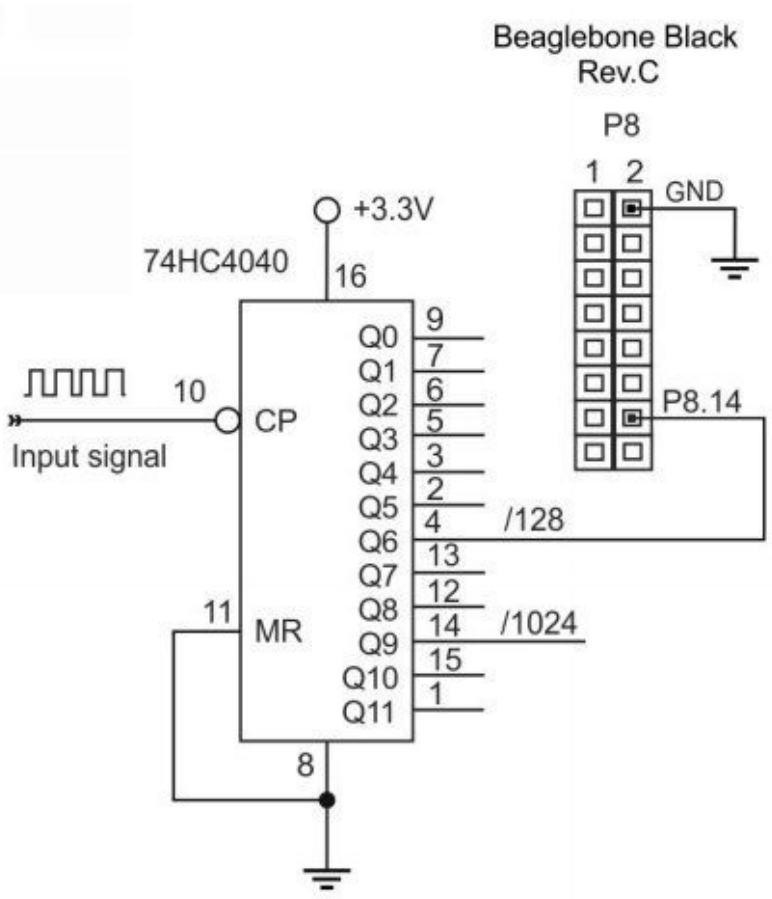


Fig.29

The input signal goes to the clock input (CP); the counter advances on the HIGH-to-LOW transition of CP. The output signals appear on twelve parallel outputs (Q0 to Q11). The high level when brought to an asynchronous master reset input (MR) clears all counter stages and pulls all outputs low, regardless of the state of CP.

The Python source code for this project is shown in **Listing 13**.

Listing 13.

```
import Adafruit_BBIO.GPIO as GPIO

import threading
```

```
import time
import math

fCount = 0
Done = 0
total = 0



GPIO.setup(“P8_14”, GPIO.IN)

GPIO.add_event_detect(“P8_14”, GPIO.BOTH)

class CountThread(threading.Thread):
    def run(self):
    global fCount
    global Done
    fCount = 0
    while Done == 0:
    if GPIO.event_detected(“P8_14”):
    fCount = fCount + 1

for cnt in range (0,19):
    Done = 0
    a = CountThread()
    a.start()
    time.sleep(1)
    Done = 1
    a.join()
    total = total + math.pow(fCount/2, 2)

total = math.sqrt(total/20)
```

```
total = round(total*1.035)
```



```
# Multiply by 128 because the input frequency is divided by 74HC4040


total = total * 128
print “Input Frequency, Hz: ” + str(total)
```

This source code is similar to that from Listing 9, but a single statement was added:

```
total = total * 128
```

This reflects the fact that the input frequency was divided by 128, so we need to multiply the result obtained by 128.

Pulse width measurement using the PWM-to-DAC converter LTC2645

Measuring the pulse width of digital input signals with the BeagleBone Black can easily be implemented by taking the LTC2645 IC from Linear Technology. Once an input frequency is known, we can calculate the pulse width of an input signal by measuring the corresponding voltage level at the LTC2645 DAC output.

A few words about this chip. The LT C2645 is a family of quad 12-, 10-, and 8-bit PWM – to – Voltage output DACs with an integrated high accuracy, low drift and 10ppm/°C reference. It has rail-to-rail output buffers and is guaranteed monotonic.

The LTC2645 measures the period and pulse width of the PWM input signals and updates the voltage output DACs after each corresponding PWM input rising edge. The DAC outputs update and settle to 12-bit accuracy within 8µs typically and are capable of sourcing and sinking up to 5mA (3V) or 10mA (5V), eliminating voltage ripple and replacing slow analog filters and buffer amplifiers.

The LTC2645 has a full-scale output of 2.5V using the 10ppm/°C internal reference. It can operate with an external reference, which sets the full-scale output equal to the external reference voltage.

Each DAC enters a pin-selectable idle state when the PWM input is held unchanged for more than 60ms.The part operates from a single 2.7V to 5.5V supply and supports PWM input voltages from

1.71V to 5.5V.

The hardware circuit for this project is shown in **Fig.30**.

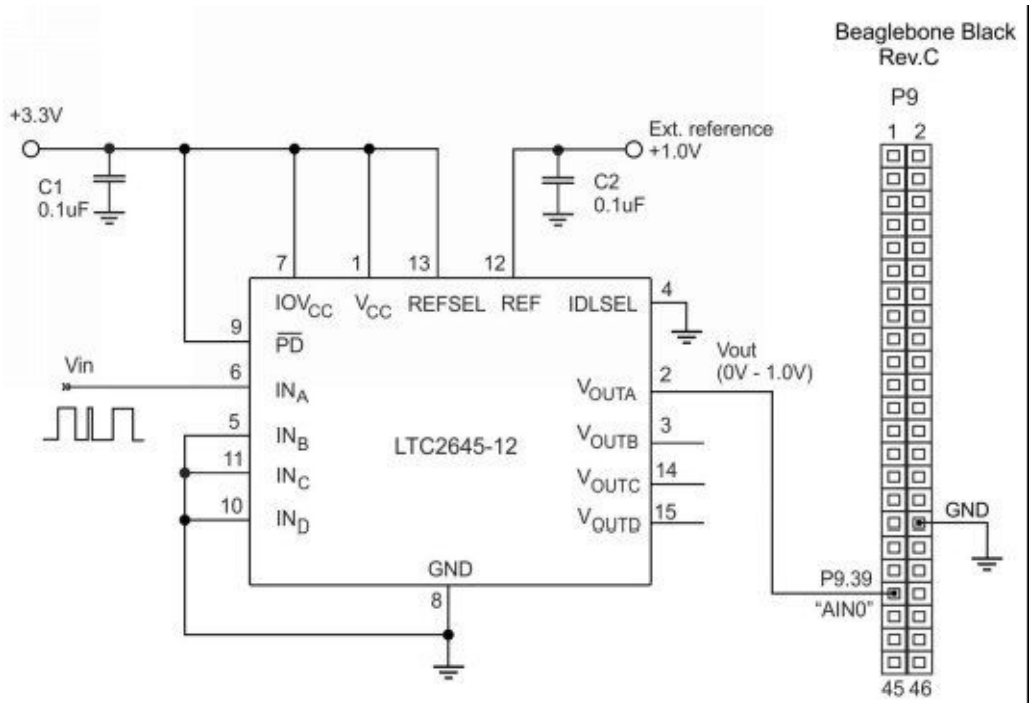


Fig.30

Inputs $\mathbf{IN_A}$, $\mathbf{IN_B}$, $\mathbf{IN_C}$, $\mathbf{IN_D}$ (pins 6, 5, 11 and 10) are PWM Inputs. These pins should be fed by a pulse-width modulated input frequency between 30Hz and 6.25 KHz (12-bit), 25 KHz (10-bit) or 100 KHz (8-bit).

After a rising edge arrives at $\mathbf{IN_X}$ pin, the device calculates the duty cycle based upon the pulse width and period and updates a corresponding DAC channel $\mathbf{V_{OUTX}}$.

The voltage level on **REFSEL** (pin 13) determines the reference mode. By connecting this pin to "ground", we select the internal reference mode. To select the external reference mode we need to connect **REFSEL** to the positive power rail $\mathbf{V_{CC}}$. In this circuit, the external reference mode was chosen.

The **REF** (Pin 12) is the reference voltage input or output. When **REFSEL** is connected to $\mathbf{V_{CC}}$, **REF** is an input ($1V \le V \le V_{CC}$) where the voltage supplied sets the full-scale DAC output voltage. When **REFSEL** is connected to "ground", the 10ppm/°C, 1.25V internal reference (half full-scale) is available at the pin. This output may be bypassed to "ground" with up to 10µF and must be buffered when driving an external DC load current.

The DAC output voltage $\mathbf{V_{OUTX}}$ can be calculated by the following equation:

$V_{OUTX} = VREF \times (t_{PWHX} / t_{PERX})$ (1),

where **VREF** is 2.5V in internal reference mode or the **REF** pin voltage in external reference mode, $\mathbf{t_{PWHX}}$ is the pulse width of the preceding $\mathbf{IN_X}$ period and $\mathbf{t_{PERX}}$ is the time between the two most recent $\mathbf{IN_X}$ rising edges.

In this circuit, the input digital signal is taken to the channel A (pin $\mathbf{IN_A}$) of LTC2645. The output analog signal on pin $\mathbf{V_{OUTA}}$ is fed to the AIN0 channel (pin **P9.39**) of the BeagleBone Black. Assuming that we choose the $\mathbf{V_{OUTA}}$ voltage is between 0V and 1V, we should connect pin **REFSEL** to the +3.3V power rail and the external reference voltage 1.0V was fed to the **REF** pin of LTC2645.

Assuming the input frequency is 1000 Hz, we can calculate the pulse width using the following source code (**Listing 14**).

Listing 14.

```
import Adafruit_BBIO.ADC as ADC
import time

ADC.setup()
AIN0 = “P9_39”
Vref = 1.0                  # external reference 1.0V
bFreq = 1000.0              # base frequency 1000 Hz
tper = bFreq / 1000.0   # period = 1/f

Vin = ADC.read(AIN0) * 1.8
pW = (Vin / Vref) * tper * 1000          # period is taken in mS
print “Input voltage on AIN0: ” + str(round(Vin, 3)) + ” V”
print “Pulse Width: ” + str(round(pW,3)) + ” mS”
```

Here the **Vref** variable determines the value of the reference (1.0V, in this example); the **bFreq** is assigned the value of the base frequency of the PWM signal. The **tper** variable holds the period of the input signal. The analog signal measured on the AIN0 input will be used for calculating the pulse width of the digital input signal. This is done by the following statement:

pW = (Vin / Vref) * tper * 1000

It is seen that the above expression is based upon the formula (1).

You can easily modify the above source code for different frequencies and/or references.

The circuit shown in **Fig.30** can easily be adjusted to operate in circuits where power isolation is required. The next circuit diagram illustrates using the LTC2645 with the 6N137 optoisolator IC (**Fig.31**).

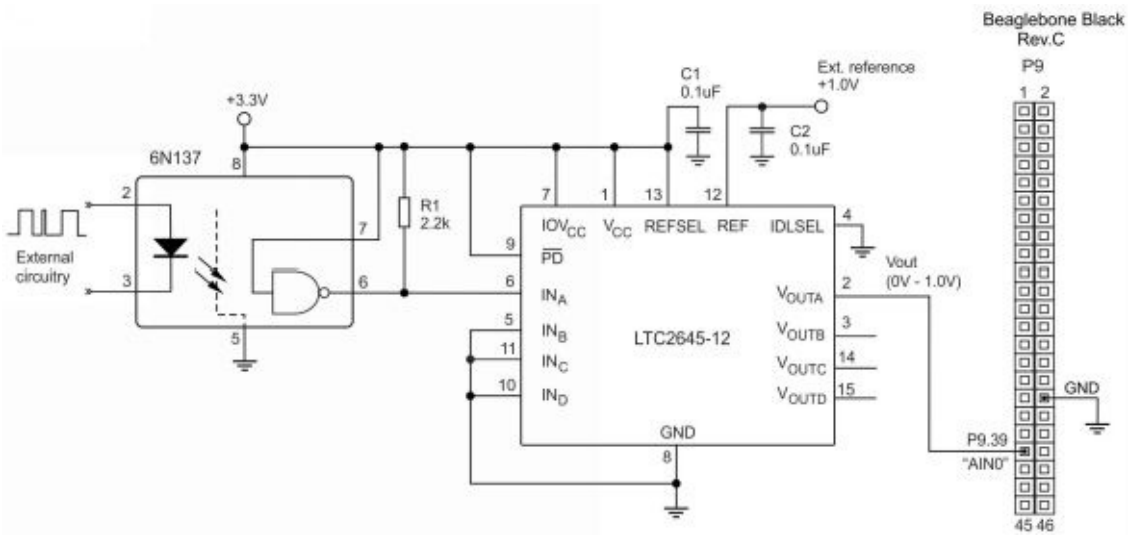


Fig.31

In this circuit, the input PWM signal from external circuitry arrives on pins 2 – 3 of the LED of 6N137 optoisolator. The TTL-compatible output is taken from pin 6 of the IC. 6N137 can be replaced by almost any similar part providing TTL-compatible output signal.

Simple control systems

The following projects illustrate using the GPIO pins of the BeagleBone Black in simple measurement and control systems.

The next project represents the simple control system where both input and output digital signals are processed through the digital buffers. The hardware circuit of the project is shown in **Fig.32**.

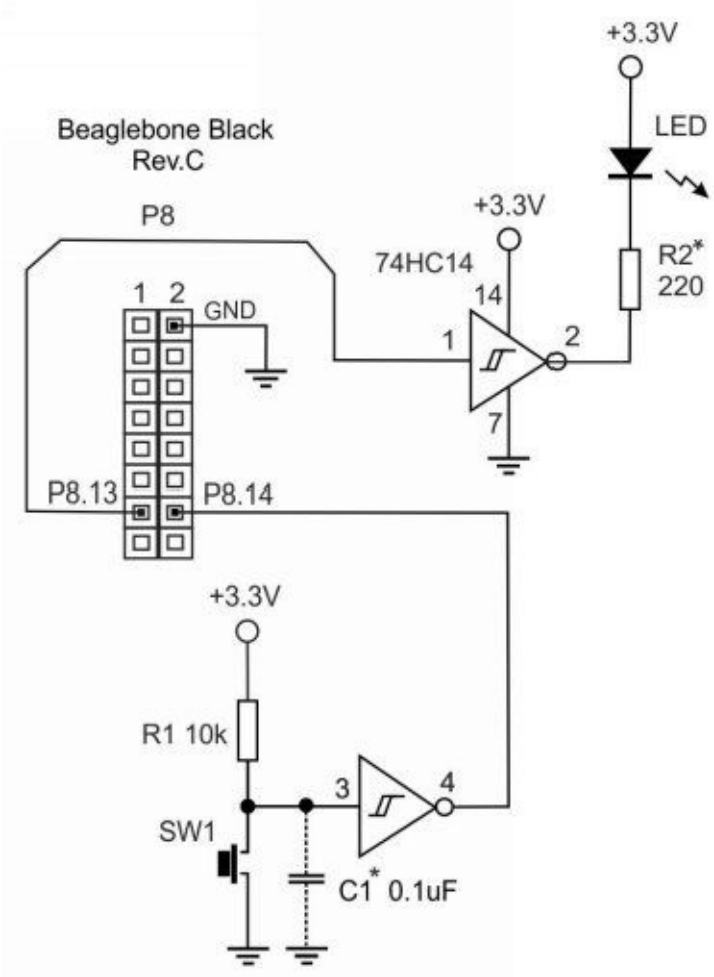


Fig.32

In this system, the program code reads the state of the switch **SW1** on pin 3 of 74HC14 buffer every 2 seconds. After pressing **SW1**, the LED on pin 2 of 74HC14 is driven ON. Conversely, when **SW1** is open, the LED is driven OFF.

The Python source code driving this circuit is shown in **Listing 15**.

Listing 15.

```
import Adafruit_BBIO.GPIO as GPIO
import time

GPIO.setup("P8_13", GPIO.OUT)
GPIO.setup("P8_14", GPIO.IN)

while (True):
    bVal = GPIO.input("P8_14")
    if (bVal != 0x0):
    print "'SW1 is pressed. LED is turned ON"
    GPIO.output("P8_13", GPIO.HIGH)
    else:
    print "SW1 is released. LED is turned OFF"
    GPIO.output("P8_14", GPIO.LOW)
    time.sleep(2)
```

The next to discuss is the system which simply drives the LED ON /OFF as the visible light intensity changes. The input signal is originated by a Light–Dependent Resistor (LDR) of 10k whose resistance changes with illumination.

The circuit diagram of such a system is shown in **Fig.33.**

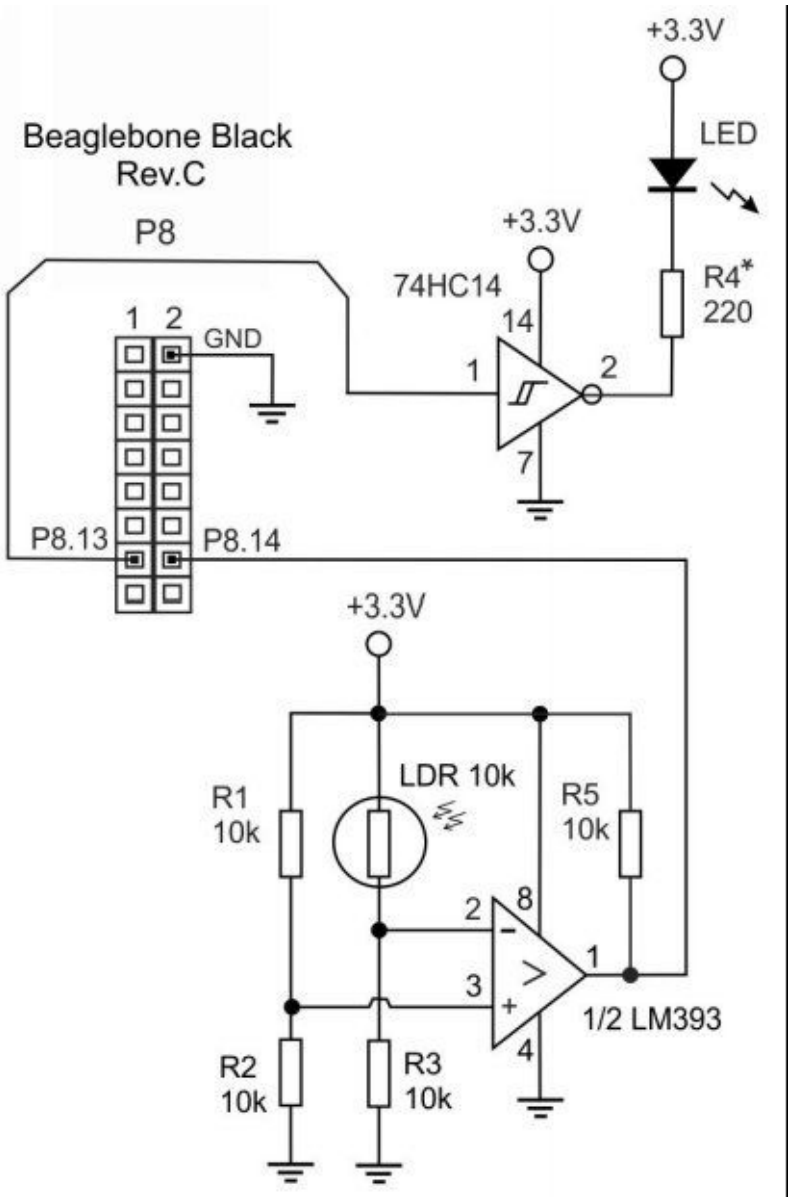


Fig.33

In this circuit, the LDR and the resistor R3 network are arranged in the voltage divider. The voltage drop across R3 fed to the inverting input of the comparator LM393 is compared with the voltage fed to the non-inverting input (pin 3) from the resistive divider R1–R2. The values of R1-R2 determine the threshold (1.65V, in our case). When light reaches the LDR, it reduces the resistance, so the voltage level on the inverting input (pin 2) of the comparator rises above 1.65V thus pulling the comparator output (pin 1) low. Conversely, when LDR is shadowed, the voltage on pin 2 drops below 1.65V, so the comparator output goes high.

The project also shows how to process analog signals in the digital domain using voltage comparators. In those cases, we don’t need an analog-to-digital converter. Additionally, the level of input signals on the comparator input may vary in a relatively wide range.

The LM393 comparator can be replaced by any general-purpose single-supply device, for example, TLC3702, LMC7211, LMC7221, etc. For the comparators with push-pull output the resistor R5 is not needed.

The program code periodically checks the voltage level on pin **P8.14**. The LED is driven OFF when **P8.14** goes low; the LED is driven ON when **P8.14** goes high.

The Python source code handling this system is given in **Listing 16**.

Listing 16.

```
import Adafruit_BBIO.GPIO as GPIO
import time

GPIO.setup(“P8_13”, GPIO.OUT)
GPIO.setup(“P8_14”, GPIO.IN)

while (True):
    bVal = GPIO.input(“P8_14”)
    if (bVal == 0x0):
    print “Light intensity increased. LED is OFF”
    GPIO.output(“P8_13”, GPIO.LOW)
    else:
    print “Light intensity dropped.LED is ON”
    GPIO.output(“P8_13”, GPIO.HIGH)
    time.sleep(2)
```

One more project describes a simple temperature control system. Compared to the previous project, the LDR – R3 network is replaced with the temperature sensor LM35. The LM35 output is connected to the inverting input of the comparator, while the reference voltage (threshold) on the non-inverting input is taken from the voltage divider R1-R2.

The source code for this system could almost be the same as that in **Listing 16**. The circuit diagram of the temperature control system is shown in **Fig.34**.

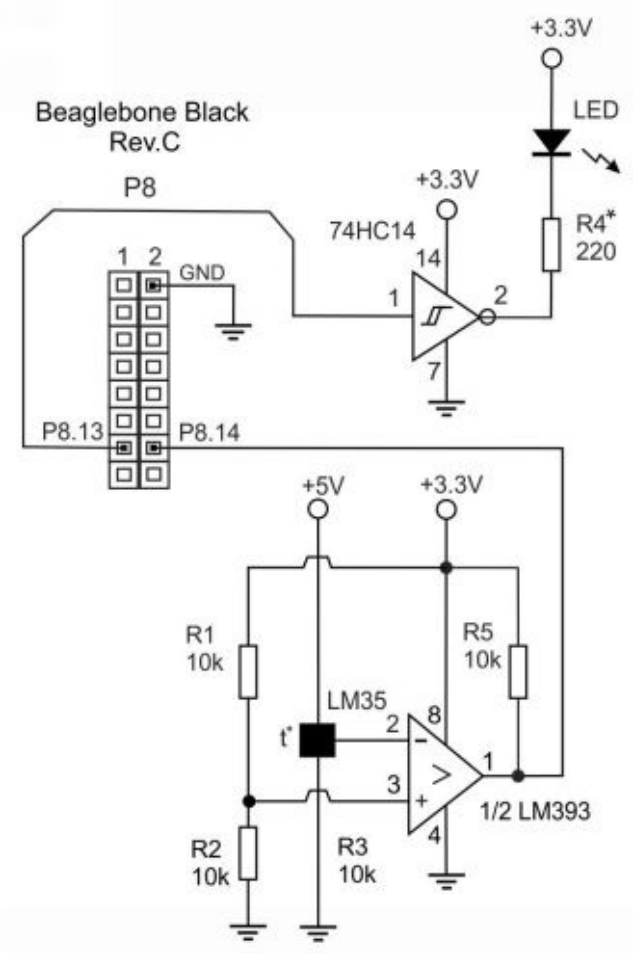


Fig.34

The resistor divider R1–R2 provides the reference voltage 0.25 V on pin 3 of the comparator that corresponds to the temperature of 25ºC. As the temperature rises above 25ºC, the LM35 output exceeds 0.25V thus causing the comparator's output to go low. Conversely, as the temperature drops below 25ºC, the sensor output falls below 0.25V, so the comparator's output goes high.

The LED is driven ON/OFF depending on the comparator's output.

Generating PWM signals

Besides turning ON/OFF digital outputs the BeagleBone Black can generate the pulse trains with the programmable frequency and duty cycle. This type of signal is known as Pulse-Width Modulation (PWM). There are 8 PWM channels that can be used on the BeagleBone and each channel can be connected to a few different pins listed below:

P9.22 – P9.31 (PWM channel 0A)

P9.21 – P9.29 (PWM channel 0B)

P8.36 – P9.14 (PWM channel 1A)

P8.34 – P9.16 (PWM channel 1B)

P8.19 –P8.45 (PWM channel 2A)

P8.13 – P8.46 (PWM channel 2B)

P9.28 (PWM2)

P9.42 (PWM0)

We don't need to refer to the PWM channel by its name since the Adafruit's BeagleBone IO Python Library allows to change the PWM on each pin by referring to its physical pin number. Since a few different pins share the same PWM channel, changing one will change the other pins that share that channel.

To control PWM signals a few functions from the Adafruit_BBIO.PWM module can be applied. First, we need to link the PWM module to the Python source code by placing the directive:

```
import Adafruit_BBIO.PWM as PWM
```

Then the library functions from this module can be fetched by a program. Let's look at a few examples of using the functions from the Adafruit_BBIO.PWM module. The function

```
PWM.start(channel, duty, freq, polarity)
```

determines all parameters of a PWM signal and allows the PWM signal to be generated.

For instance, the statement

PWM.start(“P8_13”, 75, 500, 0)

activates the PWM pulse train with the base frequency of 500 Hz, the duty cycle 75 and a positive polarity on pin **P8.13**. Note that the valid values for a duty cycle are between 0.0 and 100.0.

It is also possible to redefine the parameters of the PWM signal on the fly by using a couple of functions from the Adafruit_BBIO.PWM module. The following statement changes the frequency of a PWM signal on pin **P8.13** to 1000 Hz:

PWM.set_frequenc y(“P8_13”, 1000)

To change the duty we can apply the **set_duty_cycle()** function. The following example illustrates how to set the duty of a PWM signal to 30:

PWM.set_duty_cycle(“P8_13”, 30)

If we don’t need a PWN signal anymore, we can either disable or cleanup the specific channel:

PWM.stop(“P8_13”)

PWM.c leanup()

The following simple example allows to control the frequency and duty cycle of the PWM signal on pin **P8.13** (**Listing 17**).

Listing 17.

import Adafruit_BBIO.PWM as PWM

import time

```
duty = float(input('Enter the duty cycle (0-100):'))
freq = input('Enter the frequency, Hz: ')
PWM.start("P8_13", duty, freq, 0)
time.sleep(30)
PWM.stop("P8_13")
PWM.cleanup()
```


With this code you can set the frequency and the duty of a PWM signal using a keyboard. The PWM channel remains active during 30 s.

Simple PWM-to-DC converter

The PWM output can be easily converted into DC voltage using the simplest active low-pass filter. We will discuss active filters a bit later, but just now we will employ a simple active low-pass filter to obtain the steady DC voltage from a PWM signal.

First, look at how to get DC voltage from PWM. The following diagram (**Fig.35**) illustrates the theory of operation.

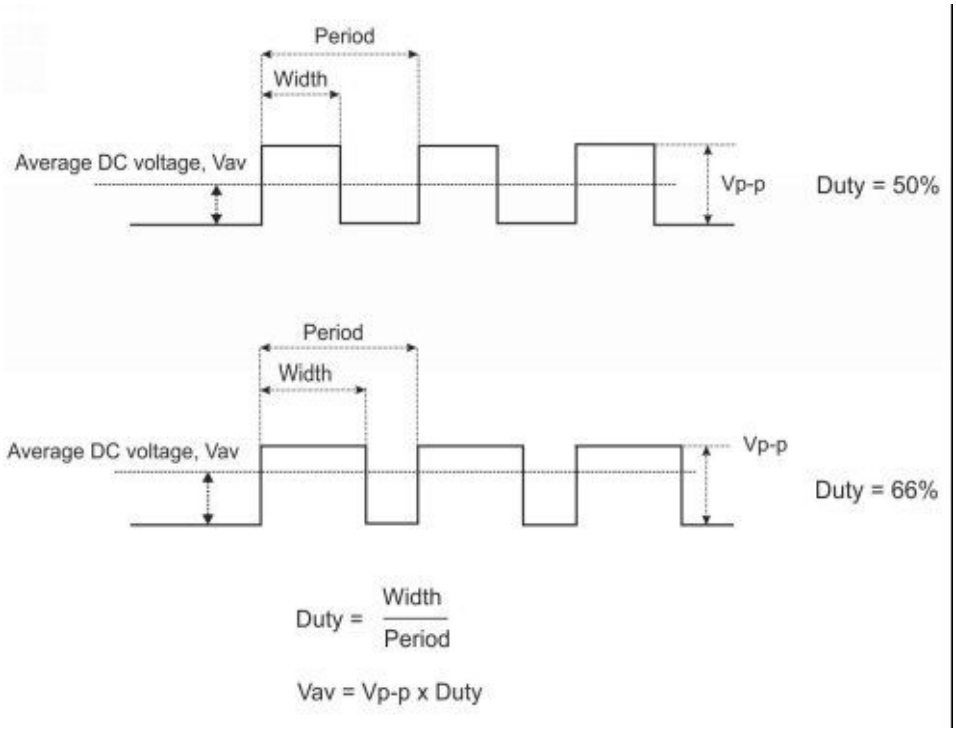


Fig.35

It is seen that the average DC voltage ($\mathbf{V_{av}}$) produced by the PWM signal is related to the peak voltage ($\mathbf{V_{p\text{-}p}}$) as:

$V_{av} = V_{p\text{-}p}$ x Duty,

where **Duty** (duty cycle) is the ratio of a pulse width to a period. For PWM circuits, the period of a signal is kept constant while the pulse width can change. If a pulse train has a constant amplitude $\mathbf{V_{p\text{-}p}}$, then the DC voltage $\mathbf{V_{av}}$ will only be determined by the duty cycle **Duty**. For TTL-compatible signals ranging from 0V to +3.3V the voltage $\mathbf{V_{p\text{-}p}}$ will be close to 3.3V. **Fig.35** illustrates the cases when the duty cycle is 50% ($\mathbf{V_{av}}$ = 1.65V)

and 66% ($\mathbf{V_{av}}$ = 2.18V).

In real life, DC output obtained from a PWM circuit can produce a large ripple voltage. Nevertheless, PWM circuits are frequently used for driving a wide variety of electronic circuits, including DC motors and buck-SPC converters. On the other hand, many circuits must be fed by a precision DC voltage. For example, voltage controlled oscillators (VCO) and precision amplifiers need a highly stable DC voltage source without any ripple injected.

To obtain a stable DC voltage from a PWM pulse train we need to add an active low-pass filter to the PWM output (**Fig.36**).

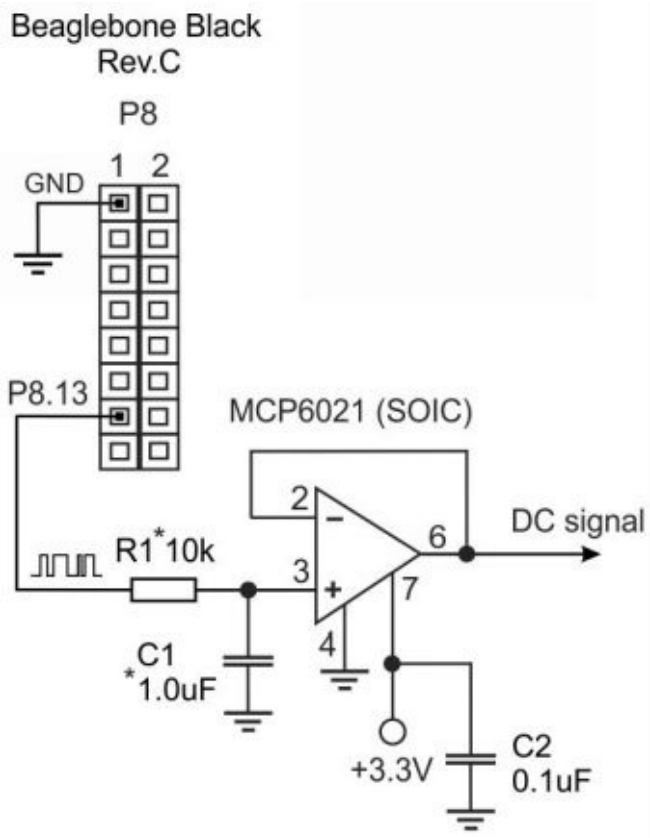


Fig.36

In this circuit, a PWM signal from pin **P8.13** goes to the R1C1 network that is a simple low-pass filter passing frequencies lower than 1 / (2 x π x R1 x C1) that is about 16 Hz, so we obtain almost a steady DC voltage on pin 6 of the op-amp MCP6021 which is arranged as a voltage follower. The voltage follower isolates the filtered circuit from a load, so the DC output voltage stays constant as the current flowing through a load changes.

Almost any single-supply op-amp providing rail-to-rail I/O capabilities and GBWP > 3MHz can fit the above circuit.

Generating high precision analog voltage using an LTC2645 device

When we need to obtain a high-precision analog output voltage which should be linearly proportional to the duty cycle of a PWM signal, we can employ the LTC2645 IC. This chip has already been discussed earlier, so let's go straight to the circuit diagram (**Fig.37)**.

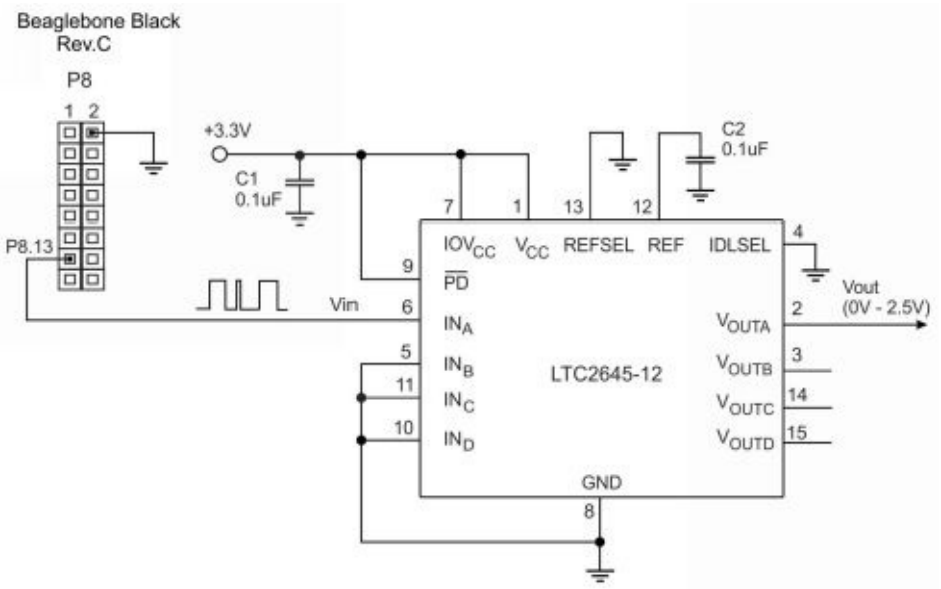


Fig.37

In our circuit, we use the internal reference 1.25V that is half full-scale (2.5V).

Remember that the DAC output voltage $\mathbf{V_{OUTX}}$ can be calculated as follows:

V_{OUTX} = VREF x (t_{PWHX} / t_{PERX}),

where **VREF** is 2.5V in internal reference mode or the voltage applied to the **REF** pin in external reference mode, $\mathbf{t_{PWHX}}$ is the pulse width of the preceding $\mathbf{IN_X}$ period and $\mathbf{t_{PERX}}$ is the time between the two most recent $\mathbf{IN_X}$ rising edges.

In this circuit, the only $\mathbf{V_{OUTA}}$ output channel is used, the rest produce zero on their outputs. If, for example, the input signal on the $\mathbf{IN_A}$ input has a frequency of 1000 Hz (period = 1 mS) and a duty cycle of 35% (0.35 off the period), then the output voltage on pin $\mathbf{V_{OUTA}}$ will be equal to 2.5 V x 0.35 = 0.875V.

Processing analog signals

So far, we've only been working with digital inputs and outputs. That is to say, only things that are on or off have been considered. For instance, the button attached to some input pin could be pressed or released, never anything in between. The same was true with outputs. The LED connected to a digital output could be either driven ON/OFF, never anything in between.

Many times, however, we deal with signals that can fall within some range of values, such as the temperature, humidity, pressure, light levels, etc. The BeagleBone Black has seven pins that can be used as analog inputs to get values from these types of sensors. Like all computers, the BeagleBone Black operates in the digital world, so signals from analog sensors must be converted to digital. The built-in circuitry of the BeagleBone that's responsible for this is called the analog-to-digital converter (A/D converter, ADC).

The ADC on the BeagleBone will let you check for continuous signals with voltage levels ranging from 0V to 1.8 V. To try out using one of the analog input pins, we will take a potentiometer which will act like a variable voltage divider. By moving the wiper of the potentiometer, we can adjust the voltage applied to the analog input pin.

You should be very careful when connecting the BeagleBone Black analog channels. First, you must use the dedicated voltage and ground pins for the A/D converter.

Remember that the input voltage on the analog pin should never exceed 1.8 V – the higher voltage level will kill your board! Don't apply a voltage to the analog pin if you feel uncertain about the range of the input signal!

To illustrate measuring an analog signal we can assemble the simplest circuit shown in **Fig.38**.

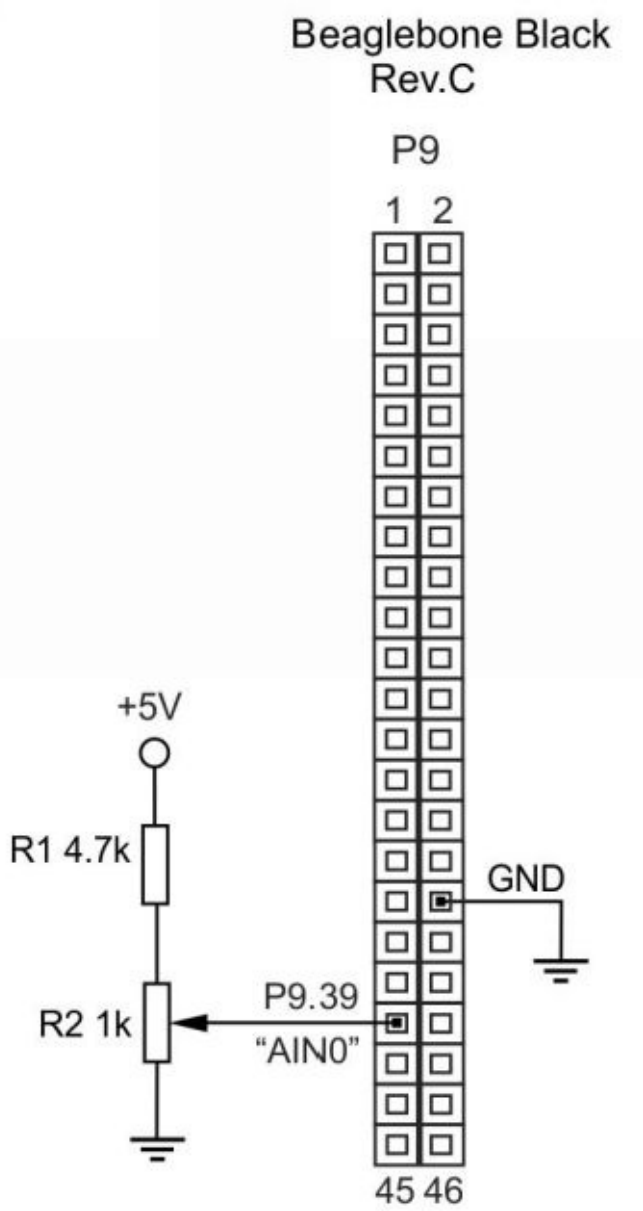


Fig.38

In this circuit, the voltage divider R1-R2 ensures that the analog input voltage fed to the channel AIN0 (pin **P9.39**) will never exceed 1 V. The test input voltage can be varied by moving the wiper of the potentiometer R2.

The simple Python source code for reading the channel AIN0 is shown in **Listing 18**.

Listing 18.

```
import Adafruit_BBIO.ADC as ADC
import time

ADC.setup()
```

```
CH0 = “P9_39”
```



```
while True:
    inpVal = ADC.read(CH0) * 1.8
    print “Input voltage on CH0: ” + str(round(inpVal, 2)) + ” V”
    time.sleep(3)
```

Here the number of channel CH0 is assigned the **P9_39** constant corresponding to AIN0. The function **ADC.read()** returns the voltage value from pin **P9.39** on a scale of 0 to 1. To get the actual voltage, we should multiply the value obtained by 1.8 using the following statement:

```
inpVal = ADC.read(CH0) * 1.8
```

The value stored in **inpVal** is then rounded to two decimal places to make it look better when it’s printed to the console. The measurements repeat every 3 s.

The application output may look like the following:

```
$ sudo python readAIN0.py

Input voltage on CH0: 0.21 V
Input voltage on CH0: 0.21 V
Input voltage on CH0: 0.21 V
Input voltage on CH0: 0.21 V
Input voltage on CH0: 0.21 V
Input voltage on CH0: 0.2 V
Input voltage on CH0: 0.19 V
Input voltage on CH0: 0.25 V
Input voltage on CH0: 0.27 V
Input voltage on CH0: 0.28 V
Input voltage on CH0: 0.29 V
```

Input voltage on CH0: 0.29 V

Input voltage on CH0: 0.29 V

Input voltage on CH0: 0.31 V

One more example presented below illustrates how to measure the temperature of environment using the LM35 temperature sensor (**Fig.39**). The sensor's output voltage **Vout** is related to the absolute temperature **T** by:

T = Vout x 100

For example, the output voltage 0.23V will correspond to 23°C. In the circuit diagram shown in **Fig.39** the output of the LM335 sensor is directly connected to the channel AIN0.

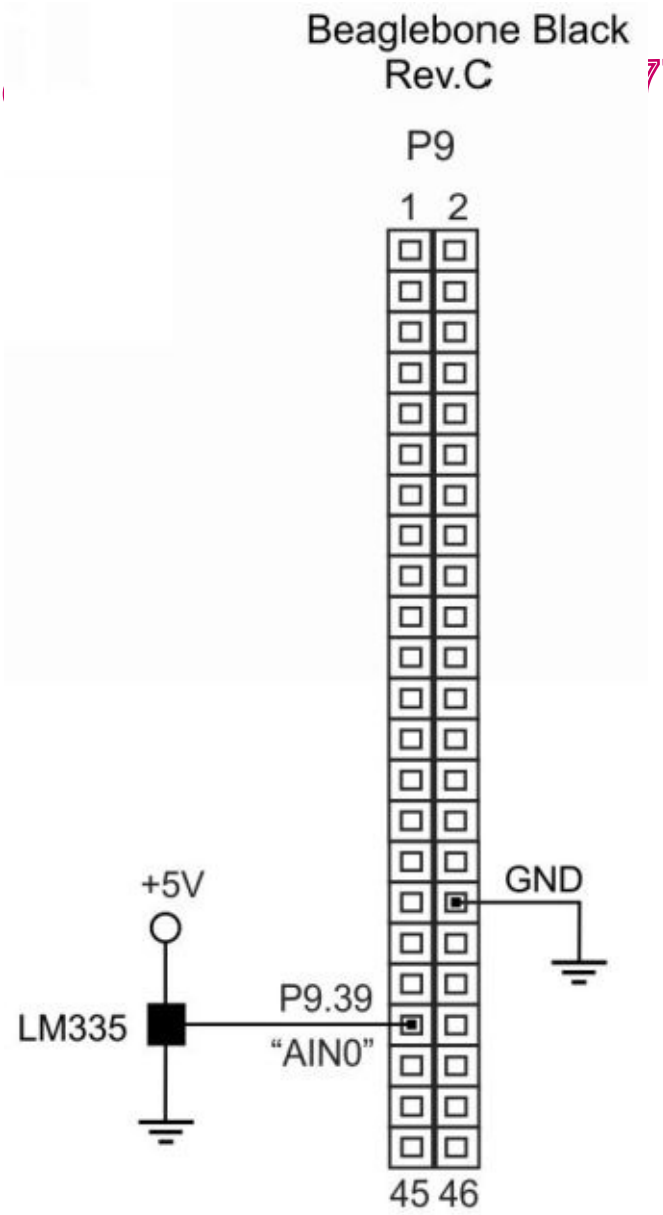


Fig.39

The Python source code for this temperature measurement system is shown in **Listing 19**.

Listing 19.

```
import Adafruit_BBIO.ADC as ADC
import time

ADC.setup()
CH0 = "P9_39"
```

```
while True:
    inpVal = ADC.read(CH0) * 1.8
    print “Input voltage on CH0: ” + str(round(inpVal, 2)) + ” V”
    print “The temperature in the room is: ” + str(round(inpVal, 2) * 100) + ” deg.Celsius”
    time.sleep(3)
```

The running application provides the following output:

```
$ sudo python readTemp.py
Input voltage on CH0: 0.17 V
The temperature in the room is: 17.0 deg.Celsius
Input voltage on CH0: 0.17 V
The temperature in the room is: 17.0 deg.Celsius
Input voltage on CH0: 0.17 V
The temperature in the room is: 17.0 deg.Celsius
Input voltage on CH0: 0.19 V
The temperature in the room is: 19.0 deg.Celsius
Input voltage on CH0: 0.18 V
The temperature in the room is: 18.0 deg.Celsius
Input voltage on CH0: 0.19 V
The temperature in the room is: 19.0 deg.Celsius
Input voltage on CH0: 0.19 V
The temperature in the room is: 19.0 deg.Celsius
Input voltage on CH0: 0.19 V
The temperature in the room is: 19.0 deg.Celsius
```

When we need to obtain precision results, we must connect a sensor to the BeagleBone through a buffer stage; this allows to avoid the signal loss because of a voltage divider formed by the sensor output and the ADC input. One possible configuration is shown in **Fig.40**.

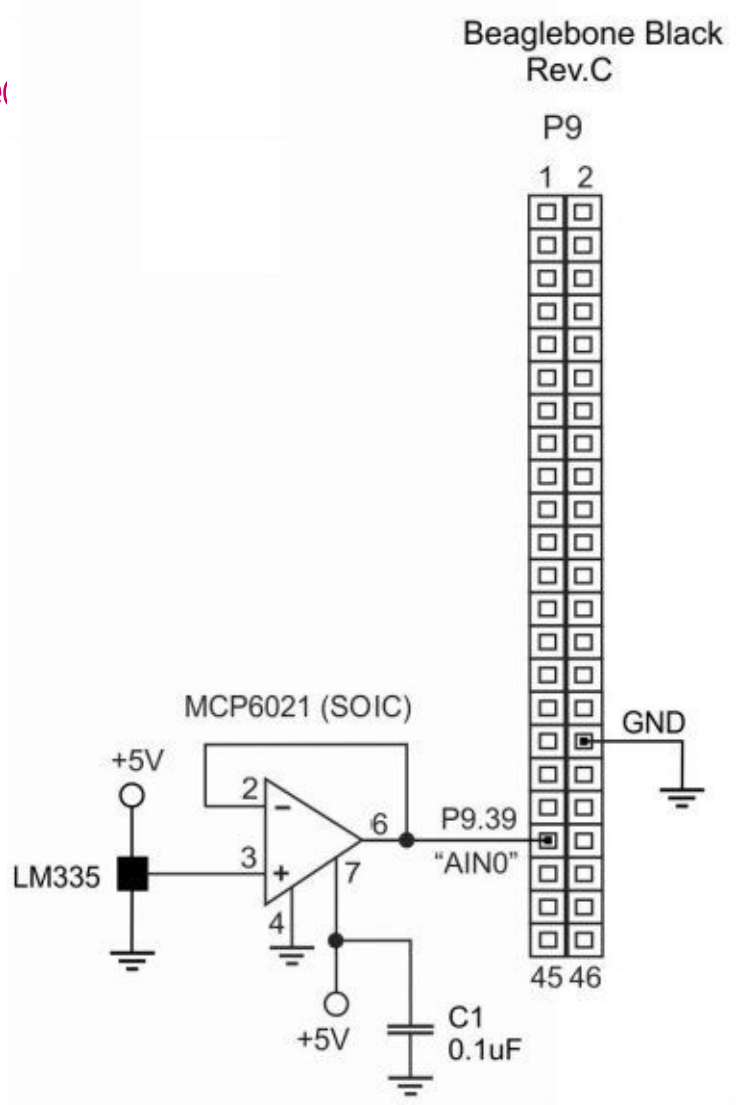


Fig.40

In this circuit, the buffer stage is arranged as a voltage follower using the op-amp MCP6021 (the pinout corresponds to the SOIC package). You can also employ any of general-purpose single supply op-amps (MCP6022, OPA364, etc.).

For analog signals beyond the allowable input range of the BeagleBone internal A/D converter we should apply a voltage divider followed by a buffer. This allows to decreases the amplitude of an input signal to the maximum safe limit. The possible configuration is shown in **Fig.41**.

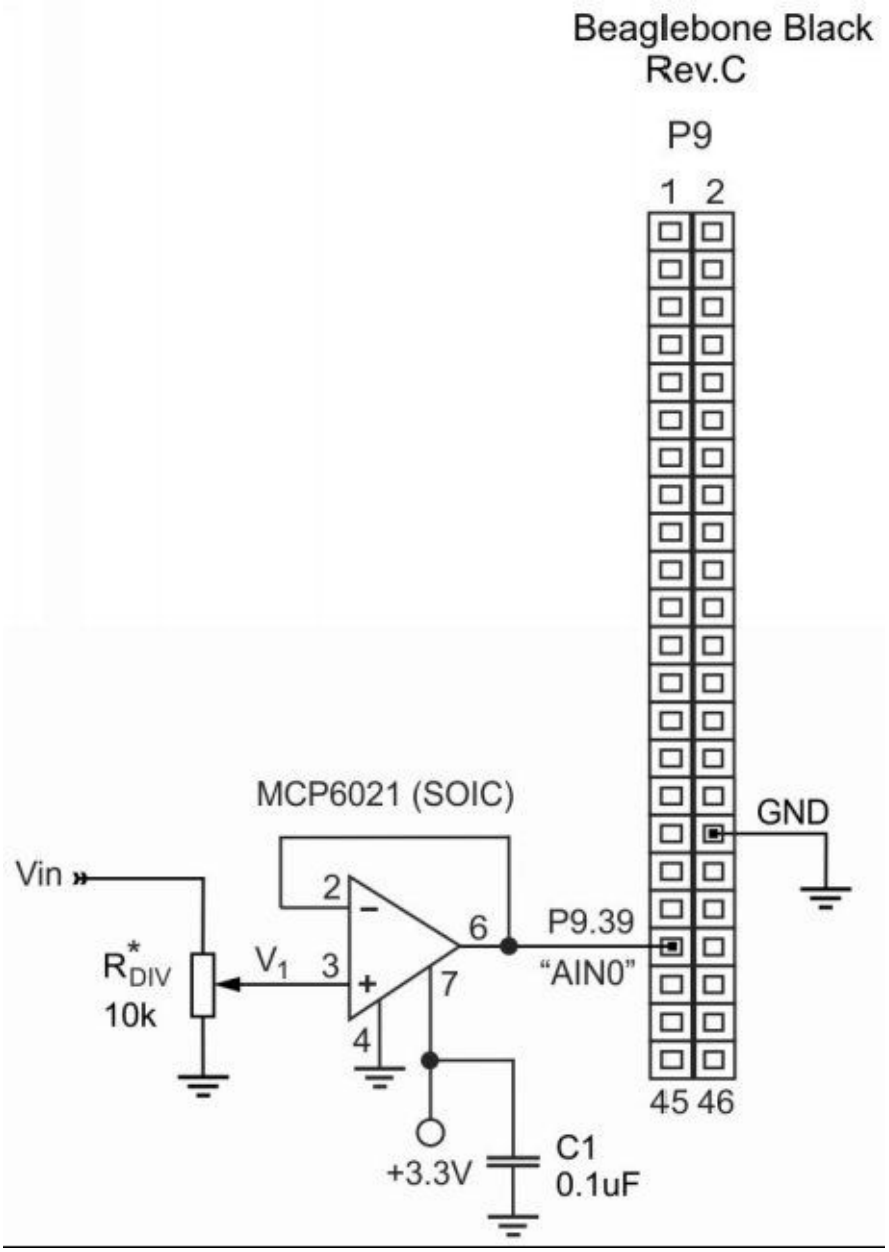


Fig.41

In this circuit, the signal $\mathbf{V_1}$ from the wiper of the potentiometer R_{DIV} goes to the buffer arranged as a voltage follower using an op-amp MCP6021. The voltage follower allows to avoid signal loss because of the divider formed by the output impedance of the source Vin and the input impedance of R_{DIV}.

Measuring analog signals with the MCP3201 A/D converter

For measuring continuous (analog) signals in a wide range we can apply some type of a stand-alone A/D converter. The popular types of ADCs used in hobby projects are SAR or Delta-Sigma devices.

This section is dedicated to using SAR (Successive Approximation Register) A/D converters. The term ″ SAR ″ refers to the device that uses the approximation technique to convert an analog input signal into a digital output code. SAR converters can typically operate in the 8- to 16-bit range and can have sample speeds up to several MSPS.

One major benefit of a SAR converter is its ability to be connected to multiplexed inputs at a high data acquisition rate. The input is sampled and held on an internal capacitor, and this charge is converted to a digital output code using the successive approximation routine. Since this charge is held throughout the conversion time, only the initial sample and hold period or acquisition time is of concern to a fast-changing input. The conversion time is the same for all conversions. This makes the SAR converter ideal for many real-time applications, including motor control, touch-screen sensing, medical and other data acquisition systems.

The next project describes programming a popular SAR analog-to-digital converter MCP3201 from Microchip Inc. This is a low-cost 12-bit single channel device driven through an SPI-compatible interface. The brief description of the device is given below (see more in the datasheet).

The Microchip MCP3201 device is a successive approximation 12-bit analog-to-digital converter with on-board sample and hold circuitry. The device provides a single pseudo-differential input. Communication with the device can be established via a simple serial SPI-compatible interface. The device can operate with sampling rates up to 100 ksps at a clock rate of 1.6 MHz. The power supply to MCP3201 device may be of 2.7V through 5.5V. Low-current design permits operation with typical standby and active currents of only 500 nA and 300 µA, respectively.

The circuit diagram of the project is shown below (**Fig.42**).

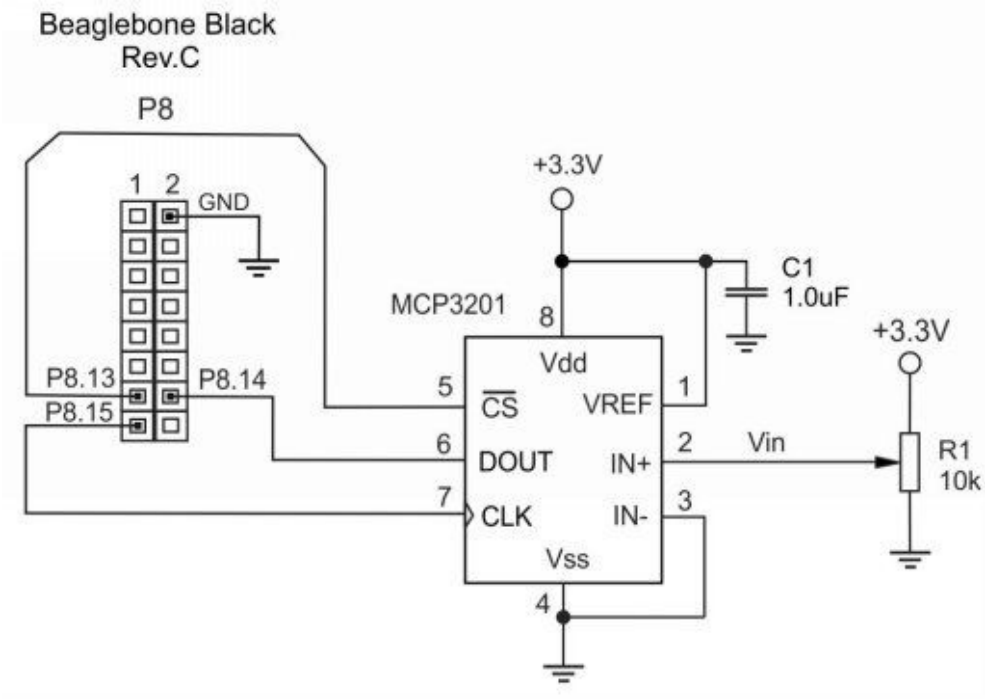


Fig.42

In this circuit, a test input voltage to the MCP3201 (pin 2) is taken from the wiper of the potentiometer R1. In real life, you can attach any sensor providing a voltage output to this pin. The connections between the BeagleBone Black and ADC MCP3201 are summarized in the following table.

BeagleBone Black P8 pin	**MCP3201 pin**
P8.15	7 (CLK)
P8.14	6 (DOUT)
P8.13	5 (CS)

One of the “ground” pins of the BeagleBone Black board must be wired to the common wire of the MCP3201 circuit. The MCP3201 device can be powered from a stand-alone +3.3V DC power supply. To simplify the circuit the voltage reference to the **VREF** input (pin 1 of MCP3201) is taken from the power rail +3.3V. To provide higher precision measurements a developer must apply a high stable reference voltage to the **VREF** input from a suitable stand-alone voltage reference IC.

The binary data stream representing the input voltage is taken in succession through the

SPI-compatible interface. The interface is driven by the **CS**, **DOUT** and **CLK** lines according to the timing diagram shown in **Fig.43**.

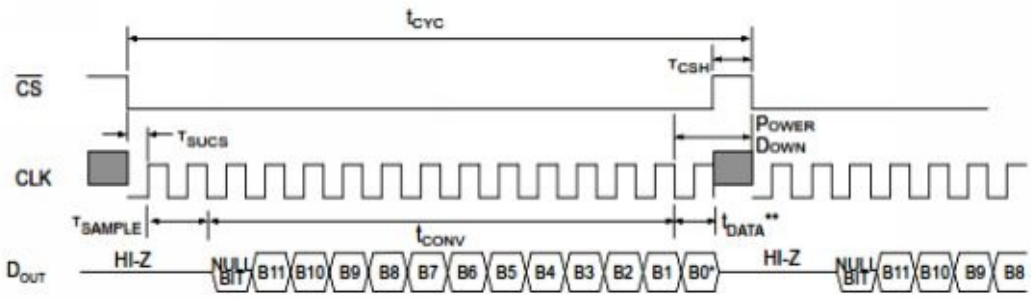


Fig.43

The single measurement cycle involves two phases. In the first phase the data sampling is executed during some time $\mathbf{T_{SAMPLE}}$. In the second phase the result of conversion (12-bit data stream) is transferred to a receiver (our BeagleBone board). The measurement cycle of the MCP3201 device begins with the **CS** signal going low. If the device was powered up with the **CS** pin low, it must be brought high and back low to initiate communication. The device will begin to sample an analog input on the first rising edge of **CLK** after **CS** is brought low.

The sample period will end on the falling edge of the second **CLK** clock, at which time a device will output a low null bit. Next 12 clocks will output the result of a conversion with MSB going first. Data is always output from a device on the falling edge of the clock; a receiver should pick up the data bit on the rising edge of the **CLK** pulse.

If all 12 data bits have been transmitted and the device continues to receive clocks while **CS** is held low, the device will output the conversion result with LSB going first. If more clocks are provided to the device while **CS** is still low (after the LSB first data has been transmitted), the device will indefinitely clock out zeros.

The reference input **VREF** is determined by the analog input voltage range and the LSB size as:

LSB size = $V_{REF} / 2^{12} = V_{REF} / 4096$

As the reference input is reduced, the LSB size is reduced as well. The theoretical digital output code produced by the A/D Converter is a function of the analog input signal and the reference input. That fact is reflected by the formula:

Digital Output Code = (V_{IN} * 4096) / V_{REF},

where:

V_{IN} – the analog input voltage between pins **IN+** and **IN-**,

V_{REF} – the reference voltage.

The conversion process is driven by the Python application whose source code is shown in **Listing 20**.

Listing 20.

```
import Adafruit_BBIO.GPIO as GPIO
import time

CS = “P8_13”
DOUT = “P8_14”
CLK = “P8_15”

Vref = 3.3   # Vref = 3.3V

GPIO.setup(CS, GPIO.OUT)
GPIO.setup(DOUT, GPIO.IN)
GPIO.setup(CLK, GPIO.OUT)

while True:
    binData = 0;
    GPIO.output(CS, GPIO.HIGH)     # CS is brough high
    GPIO.output(CLK, GPIO.HIGH)   # CLK goes high while CS is held high
    GPIO.output(CS, GPIO.LOW)      # CS goes low thus starting conversion
```

```
i1 = 14
while (i1 >= 0):
GPIO.output(CLK, GPIO.LOW)
bitDOUT = GPIO.input(DOUT)
GPIO.output(CLK, GPIO.HIGH)
bitDOUT = bitDOUT << i1
binData |= bitDOUT
i1 -= 1

GPIO.output(CS, GPIO.HIGH)
binData &= 0xFFF
inpV = Vref * binData/4096.0
print “Input voltage on the MCP3201 IN+ input: ” + str(round(inpV, 3)) + ” V”
time.sleep(5)
```

Before using the SPI interface the corresponding GPIO pins should be configured as follows:

```
GPIO.setup(CS, GPIO.OUT)
GPIO.setup(DOUT, GPIO.IN)
GPIO.setup(CLK, GPIO.OUT)
```

The measurement cycle is performed within the **while** loop. As the timing diagram implies, the full measurement cycle requires 14 clock pulses to complete. Therefore, the loop variable **i1** runs from 14 down to 0.

Before the **while** loop is entered, the conversion is enabled by bringing the **CS** line up then down. Those operations are performed by the sequence:

```
GPIO.output(CS, GPIO.HIGH)
GPIO.output(CLK, GPIO.HIGH)
GPIO.output(CS, GPIO.LOW)
```

In each iteration, the **bitDOUT** variable is assigned the current data bit taken from the **DOUT** line upon the raising edge of the **CLK** pulse. That is done by the statements:

```
GPIO.output(CLK, GPIO.LOW)
bitDOUT = GPIO.input(DOUT)
GPIO.output(CLK, GPIO.HIGH)
```

The current data bit is then shifted to the appropriate position in the **binData** variable by the following two statements:

```
bitDOUT = bitDOUT << i1
binData |= bitDOUT
```

After all bits have been transferred, the **while** loop exits and the conversion ends by bringing the **CS** line high:

```
GPIO.output(CS, GPIO.HIGH)
```

The binary code representing the analog input voltage is held in the **binData** variable. To make up the result the only lower 12 bits are taken:

```
binData &= 0xFFF;
```

Then the binary code in **binData** is converted into the float representation (variable **inpV**) and the result is output to the console:

```
inpV = Vref * binData/4096.0
```

```
print “Input voltage on the MCP3201 IN+ input: ” + str(round(inpV, 3)) + ” V”
```


Assuming that the source code is saved in the MCP3201_Test.py file, we can launch the Python application and observe the results:

```
$ sudo python MCP3201_Test.py
Input voltage on the MCP3201 IN+ input: 0.937 V
Input voltage on the MCP3201 IN+ input: 0.954 V
Input voltage on the MCP3201 IN+ input: 0.964 V
Input voltage on the MCP3201 IN+ input: 1.254 V
Input voltage on the MCP3201 IN+ input: 1.257 V
Input voltage on the MCP3201 IN+ input: 1.258 V
Input voltage on the MCP3201 IN+ input: 1.547 V
Input voltage on the MCP3201 IN+ input: 1.544 V
```

Using low-pass filters in data acquisition systems

Almost all signals regardless of their origin are contaminated with noise – that is true whether we acknowledge that fact or not. The best method to get rid of noise riding on a useful signal is to apply an analog low-pass filter (LPF). The LPF should be in every circuit where there is an analog-to-digital conversion. This is true regardless of the type of an analog-to-digital converter (ADC) used for digitizing signals.

An LPF stage must always be placed on an analog side of a circuit before signals reach the analog input of an analog-to-digital converter.

LPF allows to reduce the high frequency noise that is outside half of a sampling frequency of an analog-to-digital converter. After digitizing we can also apply a digital filter to reduce the lower in-band, frequency noise.

Besides removing the superimposed higher frequency noise from an analog signal, the LPF also eliminates extraneous noise peaks. Note that digital filters cannot eliminate such peaks. With an LPF standing prior to an A/D converter, the task of successfully achieving high-resolution is placed squarely on an analog circuit design and a converter.

The simplest analog low-pass filter can be composed of the resistor and capacitor (**Fig.44**).

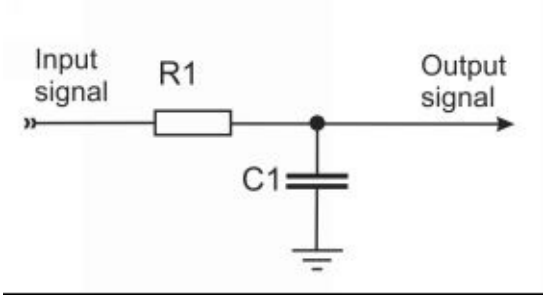


Fig.44

An input signal is passed through the R1C1pair which attenuates frequencies higher than 1/(2×π×R1×C1). This expression determines what is called the ″ cut-off ″ frequency of a low-pass filter. It is seen that the ″ cut-off ″ frequency may be set through selection of values of the resistor R1 and capacitor C1. For slow speed input signals there is a reason to set the ″ cut-off ″ frequency as low as possible, close to DC.

Adding an operational amplifier (*op-amp*) to an LPF significantly improves its characteristics. Such LPF (also called ″ active LPF ″) will match the impedances between a signal source and an input stage of ADC. The LPF can be combined with a buffer or amplifier. The simplest active analog 1st order LPF is shown in **Fig.45**.

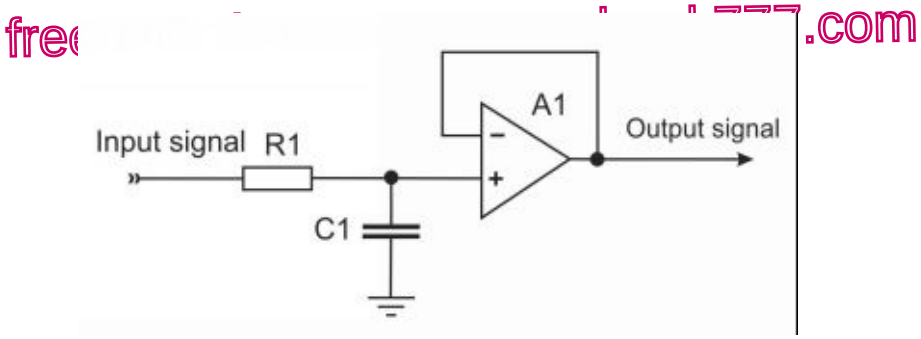


Fig.45

In this configuration, the R1C1 pair is connected to the non-inverting input of the op-amp A1 arranged as a voltage follower. Almost any single-supply op-amp may be used in such circuit (MCP601, MCP6001, MCP6021, OPA364, etc).

Remember that the output voltage of a general-purpose op-amp is usually some tens or hundreds millivolts less than the DC supply voltage, so input signals close to the positive power rail will be clipped; a similar situation occurs when signals reach 0V. A developer should take care when processing input signals close to either power rail.

To process noisy signals the circuit shown in **Fig.42** can be rearranged by introducing the 1st order active LPF prior to the A/D converter. The modified circuit is shown in **Fig.46.**

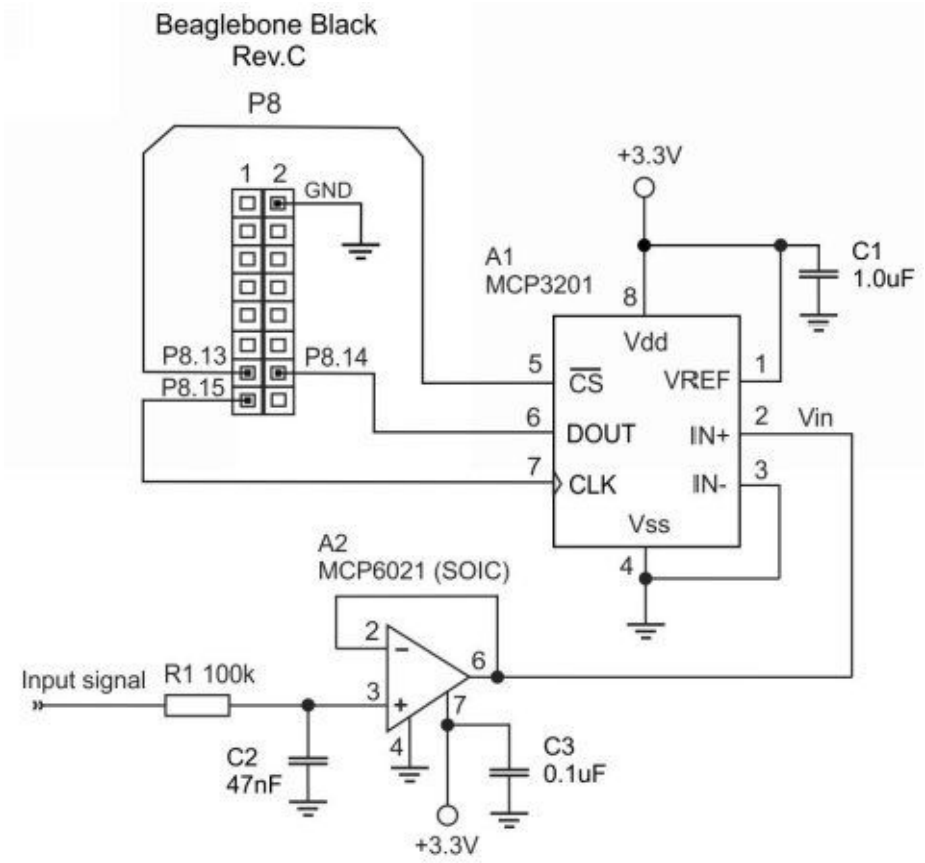


Fig.46

Here the analog input signal is passed through the active low-pass filter using the op-amp MCP6021 (A2). For the given values of R1 and C2, the ″ cut-off ″ frequency will be about 30Hz – that allows to effectively suppress high frequency noise components. Any general purpose single supply op-amp (MCP6022, OPA364, OPA348, etc.) can work in this circuit as well.

To reach the high degree of noise suppressing we can employ high-order (2^{nd} and higher) low-pass filters – you can easily find out a lot of information on that topic by surfing Internet.

The simple thermostat system using the MCP3201 A/D converter

The next project illustrates the design of a simple thermostat system which can switch on/off power to a load (cooler, for instance) depending on temperature in a room. Such system will comprise the temperature sensor connected to the analog input (pin 2 of MCP3201) and a power switch composed of a Darlington transistor and a power relay. The power switch will be driven through a GPIO pin.

Our system will check the temperature of environment using the temperature sensor LM35 that gives the output voltage proportional to absolute temperature. The relation between temperature and an output voltage appears as follows:

$$Vout = K \times T$$

where **Vout** – the voltage level on the sensor output, **T** – absolute temperature, **K** – the temperature coefficient that equals 10 mV/ºC.

This formula allows to easily calculate the temperature by measuring the LM35 output voltage. For example, the output voltage of 0.27V corresponds to the temperature 0.27/0.01 = 27ºC.

The output of LM35 is tied to the analog input of the A/D converter digitizing the analog signal and passing the result to the BeagleBone Black. Suppose that our system will turn on the cooler when the temperature of environment exceeds 30ºC. This means that the program code handling our system will constantly check if the analog input has reached the threshold (0.3V). When it occurs, the program code drives the power switch on thus connecting a power supply to a cooler. Again, when the temperature drops below 30ºC the power switch is driven off and the current flow through the cooler ceases. Our system will operate in the infinite loop providing temperature measurements every 5 seconds.

The circuit diagram of our thermostat system is given in **Fig.47**.

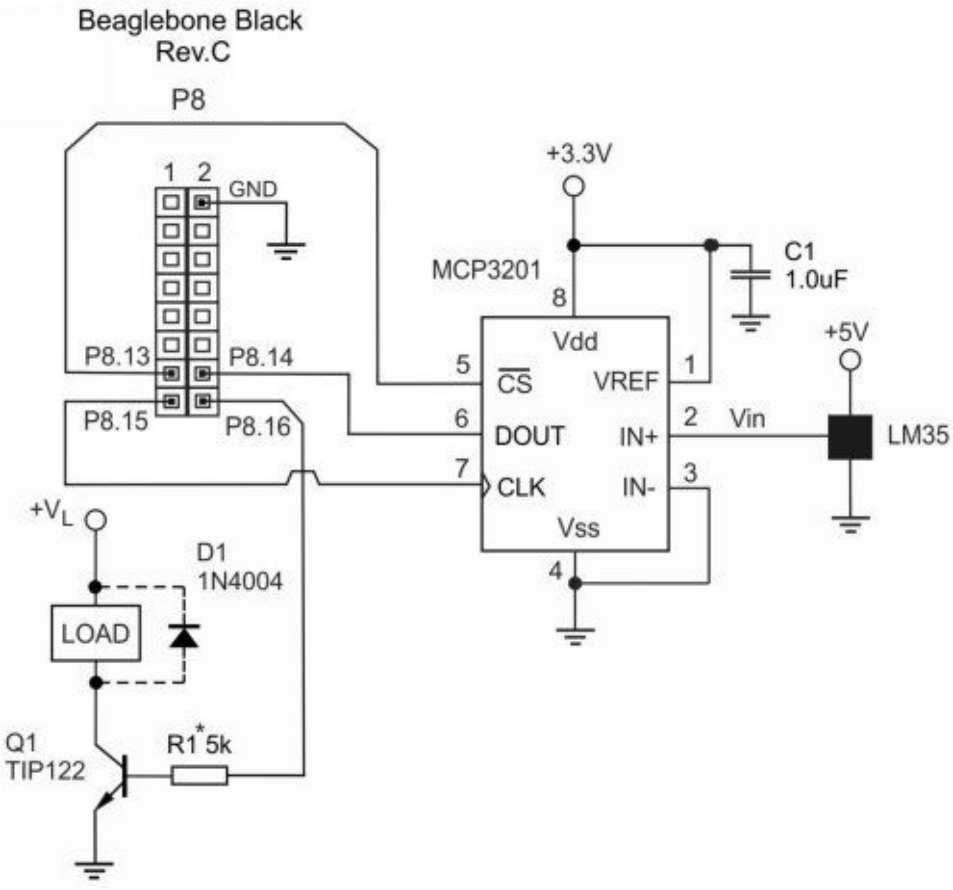


Fig.47

In this circuit, the power network (R1, Q1) is connected to the **P8.16** pin. As the **P8.16** output is pulled high (log.”1”), the Darlington transistor Q1 (TIP122) is opened by the high level (log. ″ 1 ″ or 3.3V). The LOAD is then powered from the voltage source $\mathbf{+V_L}$ through the Q1. Conversely, the low level on the **P8.16**pin (log. ″ 0 ″ or 0V) will cut off Q1 thus breaking the power chain to the LOAD. The diode D1 (indicated by the dashed line) protects the power transistor Q1 when driving an inductive load.

Some words about the LOAD. This can be either power relay with a +5V coil or a solid state relay (SSR) with the inner optoisolator circuit. The system was tested with FOTEK SSR-25DA and Omron G3NA-210B solid state relays.

The connections between the BeagleBone Black board and the external circuitry are summarized in the following table.

BeagleBone P8 pin	**MCP3201 pin**
P8.15	7 (CLK)
P8.14	6 (DOUT)
P8.13	5 (CS)

P8.16	the right end of the resistor R1

The common wire (“ground”) of the BeagleBone Black board (pin **P8.1** or **P8.2**) must be wired to the common wire of the MCP3201 circuit.

The source code handling the hardware is shown in **Listing 21**.

Listing 21.

```
import Adafruit_BBIO.GPIO as GPIO
import time

CS = “P8_13”
DOUT = “P8_14”
CLK = “P8_15”
SW = “P8_16”
Vref = 3.3   # Vref = 3.3V

GPIO.setup(CS, GPIO.OUT)
GPIO.setup(DOUT, GPIO.IN)
GPIO.setup(CLK, GPIO.OUT)
GPIO.setup(SW, GPIO.OUT)

GPIO.output(SW, GPIO.LOW)  # initially the cooler is OFF

while True:
    binData = 0;
    GPIO.output(CS, GPIO.HIGH)    # CS is brought high
    GPIO.output(CLK, GPIO.HIGH)   # CLK goes high while CS is held high
    GPIO.output(CS, GPIO.LOW)     # CS goes low thus starting conversion
```

```
i1 = 14

while (i1 >= 0):
GPIO.output(CLK, GPIO.LOW)
bitDOUT = GPIO.input(DOUT)
GPIO.output(CLK, GPIO.HIGH)
bitDOUT = bitDOUT << i1
binData |= bitDOUT
i1 -= 1

GPIO.output(CS, GPIO.HIGH)
binData &= 0xFFF
inpV = Vref * binData/4096.0
print “Input voltage on the MCP3201 IN+ input: ” + str(round(inpV, 3)) + ” V”

if (inpV >= 0.3):
GPIO.output(SW, GPIO.HIGH)
print “The cooler is ON!”
else:
GPIO.output(SW, GPIO.LOW)
print “The cooler is OFF!”
time.sleep(5)
```

The program code accomplishes its task in the endless **while (True)** loop. First, the output voltage from the temperature sensor is picked up by executing the inner **while (i1 >= 0)** loop. The calculated value of the LM35 output is moved to the **inpV** variable. The **if…else** statement checks whether **inpV** has exceeded 0.3 (that corresponds to 30ºC). Once it occurs, the **P8.16** pin is driven high thus turning the power switch ON. Conversely, when the **inpV** variable contains the value less than 0.3 the power switch is driven OFF.

Processing small analog signals with instrumentation amplifiers

This section covers a common approach to processing small (low-level) analog signals in the BeagleBone Black measurement systems. Small signals are usually present in measurement systems for biology, medicine and physics – to name but a few.

High precision measurements of small signals usually require applying special devices called ″ instrumentation amplifiers ″ or, in short, in-amps.

Instrumentation amplifiers (*in-amps*) show up in a broad spectrum of applications: measuring heart signals, factory monitoring equipment, aircraft controls, and even animal tagging. Developers have found them to be a simple and effective way to amplify small signals and remove power-line noise. A typical in-amp usually has two adjustments: gain and reference voltage. Unlike op-amps, where poor feedback design means oscillation, in-amps are quite stable.

The instrumentation amplifier’s ease of use can lead to a sense of unconcern. While it is easy to get an in-amp up and running on the bench, poor attention to detail can lead to mediocre performance in the field. Since an in-amp is typically connected directly to a sensor, designers must think about the full range of signals this sensor could present.

To illustrate the concept we will discuss using two popular in-amps: INA326 from TI and AD623 from Analog Devices. When you know how those devices operate, you will easily apply any other type of numerous in-amps available. Besides using in conditioning circuits, in-amps allow to build precision current/voltage sources, analog active filters, integrators and much more useful circuits.

Let’s begin with the INA326 device.

Fig.48 shows the basic connections required for operation of the INA326 in-amp. A bypass capacitor of 0.1µF placed close to and across the power-supply pins is strongly recommended for the highest accuracy.

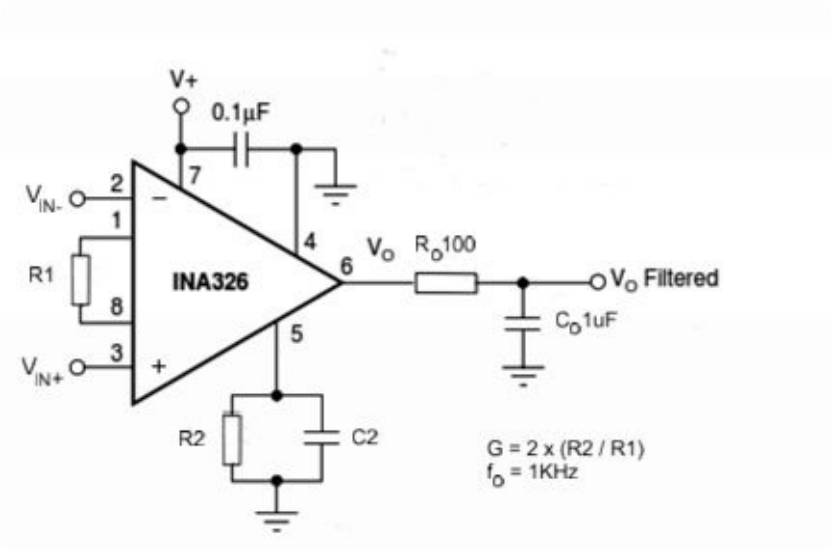


Fig.48

Here **Vin+** and **Vin-**are pins where the differential input signal arrives. For measuring positive signals we can connect the inverting input (pin 2) of INA326 to the ″ ground ″ . Note that the single-supply operation may require R2 > 100k for the full output swing. This may produce the higher input referred offset voltage. The output filter R_0C_0 serves as an anti-aliasing filter prior to an A/D converter. The voltage gain **G** of this circuit is determined as

$G = 2 \times (R2/R1)$

The table below (**Fig.49**) lists the calculated values for the gain **G** at different values of R1, R2 and C2.

DESIRED GAIN	R_1 (Ω)	$R_2 \| C_2$ (Ω || nF)
0.1	400k	20k || 5
0.2	400k	40k || 2.5
0.5	400k	100k || 1
1	400k	200k || 0.5
2	200k	200k || 0.5
5	80k	200k || 0.5
10	40k	200k || 0.5
20	20k	200k || 0.5
50	8k	200k || 0.5
100	4k	200k || 0.5
200	2k	200k || 0.5
500	2k	500k || 0.2
1000	2k	1M || 0.1
2000	2k	2M || 0.05
5000	2k	5M || 0.02
10000	2k	10M || 0.01

Fig.49

The circuit diagram of the measurement system processing low-level analog signals with the INA326 in-amp will look like that shown in **Fig.50.**

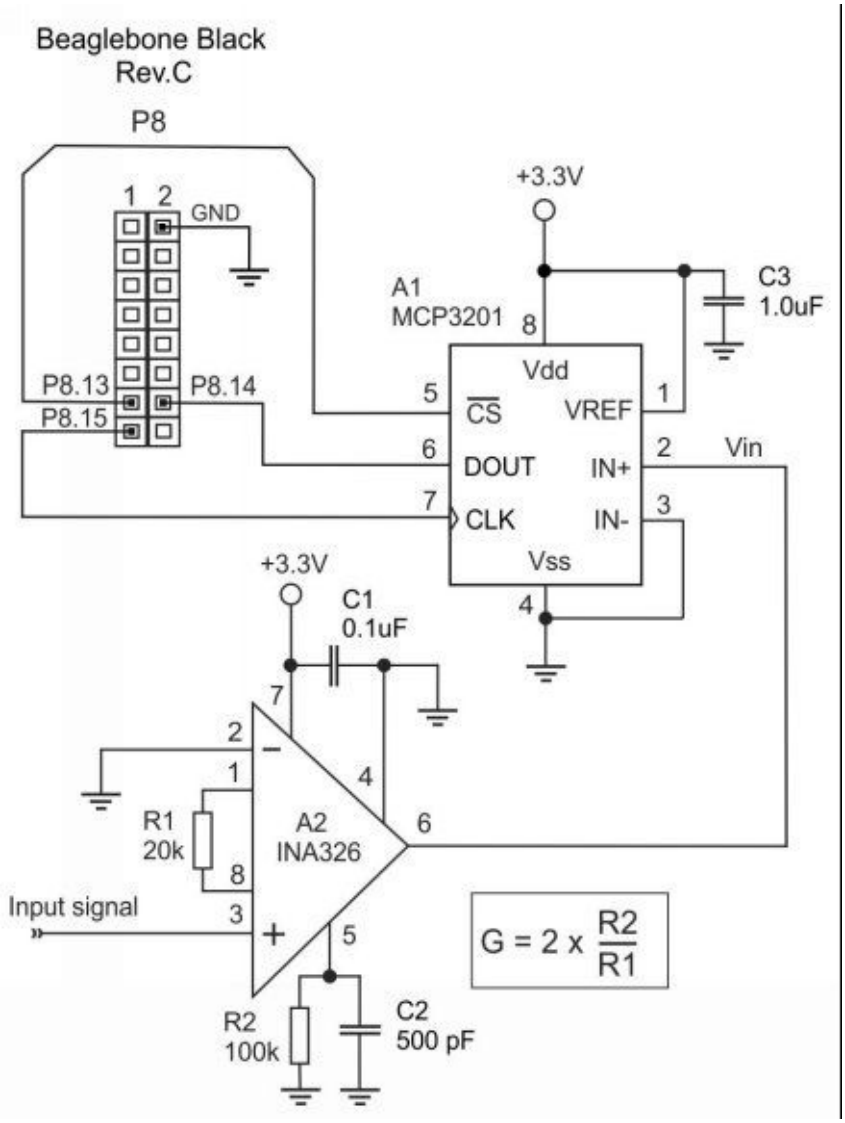


Fig.50

For the given R1 and R2, the gain **G** of the amplifier A2 will be equal to 10. This fact will be reflected in the Python application when calculating the value of the input voltage on pin 3 of A2. The Python source code for this system is shown in **Listing 22**.

Listing 22.

```
import Adafruit_BBIO.GPIO as GPIO
import time

CS = “P8_13”
DOUT = “P8_14”
CLK = “P8_15”
```

```
Vref = 3.3   # Vref = 3.3V
```



```
Gain = 2 * (98.7 / 20)       # Vout = G * Vin = 2 * (R2 / R1) * Vin

GPIO.setup(CS, GPIO.OUT)
GPIO.setup(DOUT, GPIO.IN)
GPIO.setup(CLK, GPIO.OUT)

while True:
    binData = 0;
    GPIO.output(CS, GPIO.HIGH)    # CS is brought high
    GPIO.output(CLK, GPIO.HIGH)   # CLK goes high while CS is held high
    GPIO.output(CS, GPIO.LOW)     # CS goes low thus starting conversion

    i1 = 14

    while (i1 >= 0):
    GPIO.output(CLK, GPIO.LOW)
    bitDOUT = GPIO.input(DOUT)
    GPIO.output(CLK, GPIO.HIGH)
    bitDOUT = bitDOUT << i1
    binData |= bitDOUT
    i1 -= 1

    GPIO.output(CS, GPIO.HIGH)
    binData &= 0xFFF
    inpV = Vref * binData/4096.0
    inpV = inpV / Gain
    print “Analog voltage at in-amp input = “ + str(round(inpV)) + “V”
    time.sleep(5)
```

Since the input voltage at the ADC input (pin 2 of MCP3201) is approximately 10 times greater than the voltage level at pin 3 of INA326 (**Gain** = 10), the value stored in the **inpV** variable is divided by the **Gain** constant:

inpV = inpV / Gain

Note that we need to know the exact values of resistors R1 and R2 in order to obtain the exact value of **Gain**.

Take a look at one more project where the instrumentation amplifier AD623 from Analog Devices is used. The brief description of this device taken from the datasheet is given below.

The AD623 device is an integrated single-supply instrumentation amplifier that delivers rail-to-rail output swing on a 3 V to 12 V supply. AD623 offers superior user flexibility by allowing single gain set resistor programming and by conforming to the 8-lead industry standard pinout configuration. With no external resistor, the AD623 in-amp is configured for unity gain (G = 1), and with an external resistor, AD623 can be programmed for gains up to 1000.

The AD623 device holds errors to a minimum by providing superior CMRR that increases with increasing gain. Line noise, as well as line harmonics, is rejected because the CMRR remains constant up to 200 Hz. The AD623 in-amp has a wide input common mode range and can amplify signals that have a common-mode voltage 150 mV below ground.

Although the design of AD623 was optimized to operate from a single supply, this chip still provides superior performance when powered from a dual voltage supply (±2.5 V to ±6.0 V).

The circuit diagram of the conditioning circuit with the AD623 in-amp is shown in **Fig.51**.

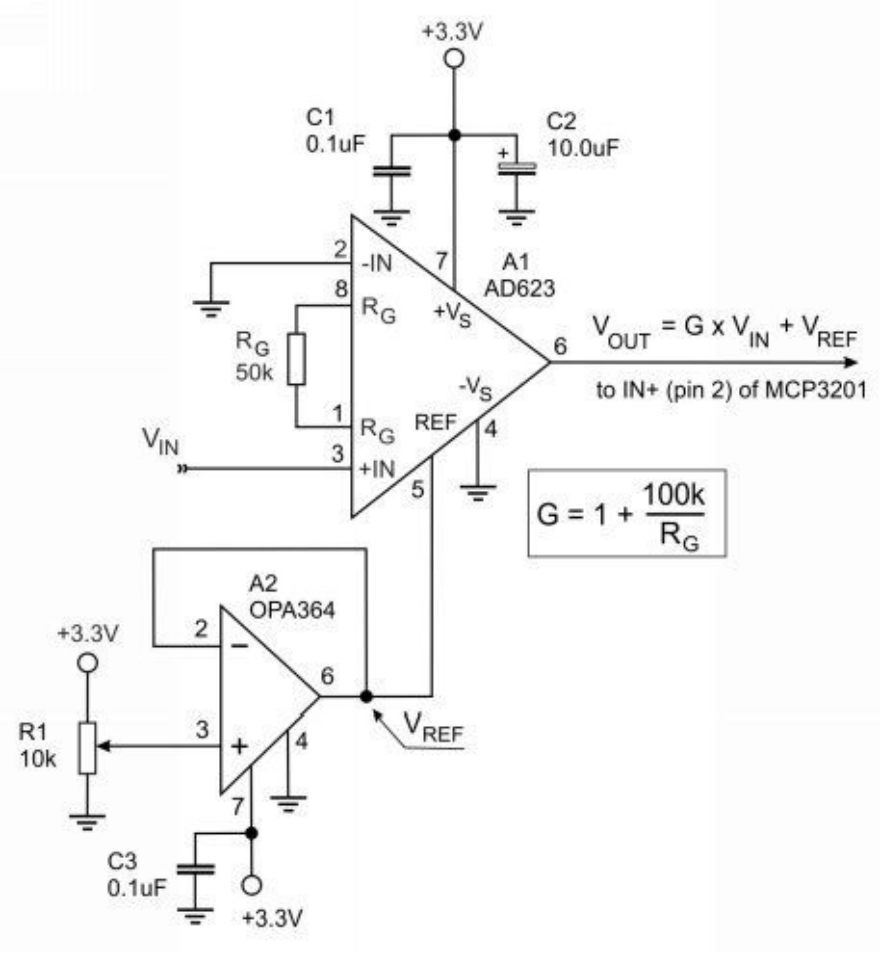


Fig.51

In this circuit, the input voltage to be measured ($\mathbf{V_{IN}}$) is fed to the **IN+** input while **IN-** is tied to ground. The amplified input voltage appears on pin 6 of A1; this voltage can be applied to the A/D converter input (pin 2 of MCP3201).

The measurement circuit using AD623 will amplify the differential signal $\mathbf{V_{dif}}$ between **+IN** and **–IN** inputs according to the following formula:

$$V_{OUT} = (1 + 100K / R_G) \times V_{dif} + V_{REF},$$

where $\mathbf{V_{out}}$ is the output voltage on pin 6 of AD623, $\mathbf{R_G}$ is the value of the resistor connected between pins 1 and 8, $\mathbf{V_{dif}}$ is the differential voltage applied between the **+IN** and **–IN** inputs and $\mathbf{V_{REF}}$ is the reference voltage applied to the **REF** input (pin 5). As it is seen from the above formula, the gain **G** of the AD623 amplifier is determined by the single resistor R_G:

$$G = 1 + 100k / R_G$$

In our case, the gain **G** is equal to 3. Since the differential input voltage $\mathbf{V_{dif}} = \mathbf{V_{IN}}\text{-}\mathbf{0}$, the output voltage $\mathbf{V_{OUT}}$ turns out to be

$$V_{OUT} = G \times V_{IN} + V_{REF}$$

The above relationships should be taken into consideration when calculating the measured input voltage within the program code.

The reference terminal **REF** (pin 5 of AD623) is assigned the potential which determines the output voltage at $\mathbf{V_{dif}} = 0$; that is especially useful when a load does not share a precise ground with the rest of the system. It provides a direct means of injecting a precise offset to the output. The reference terminal is also useful when bipolar signals are being amplified because it can be used to provide a virtual ground voltage. The voltage on the reference terminal can be varied in the full range from the negative to positive power rail.

Note that the **REF** pin should be tied to a low impedance point for optimal common-mode rejection (CMR), so don't connect the resistive divider formed by R1 directly to the **REF** pin. In our circuit, the voltage from the divider R1 is fed to pin 5 of op-amp OPA364. The voltage follower formed by the op-amp OPA364 acts as a buffer providing very low output impedance suitable for **REF**. The potentiometer R1 should be of high precision and have the impedance from a few tens of Ohms to several KOhms.

You can take any general purpose single supply op-amp (OPA348, OPA248, MCP601, MCP6021, etc.) in the voltage follower circuit. But, remember, in order to process bipolar input signals you need to select the proper **VREF** voltage so that to avoid the distortion of input signals.

Some words about powering the in-amp AD623. The power supply is connected to $+V_S$ and $-V_S$ terminals. The supply can be either bipolar (V_S = ±2.5V to ±6V) or single supply ($-V_S$ = 0V, $+V_S$ = 3.0V to 12V). Power supplies should be capacitively decoupled close to the power pins of the device. For the best results, use surface-mount 0.1 µF ceramic chip capacitors and 10 µF electrolytic tantalum capacitors.

For better understanding a device, it would be worth learning the AD623 datasheet before the in-amp is put in the circuit.

Using high-resolution Delta-Sigma analog-to-digital converters

To perform analog-to-digital conversions with very high precision we need to take some delta-sigma (also called ″ sigma-delta ″) analog-to-digital converter. The delta-sigma (Δ-Σ) ADC is the converter of choice for modern voice band, audio and high precision industrial measurement applications. The delta-sigma ADC tackles the application demands of a slow analog signal that requires a high signal-to-noise-ratio (SNR) and wide dynamic range. Delta-sigma converters are ideal for converting signals over a wide range of frequencies from DC to several MHz with very high resolution.

The signal chain for the delta-sigma converter application starts with a sensor (**Fig.52)**. Unlike circuits with SAR analog-to-digital converters, Δ-Σ devices don't require additional analog gain circuits such as amplifiers and instrumentation amplifiers, following the sensor block. Between the sensor and the delta-sigma A/D converter, there will only be an anti-aliasing, active or passive low-pass filter. As **Fig.52** illustrates, the delta-sigma ADC generally requires only a 1st order passive RC filter.

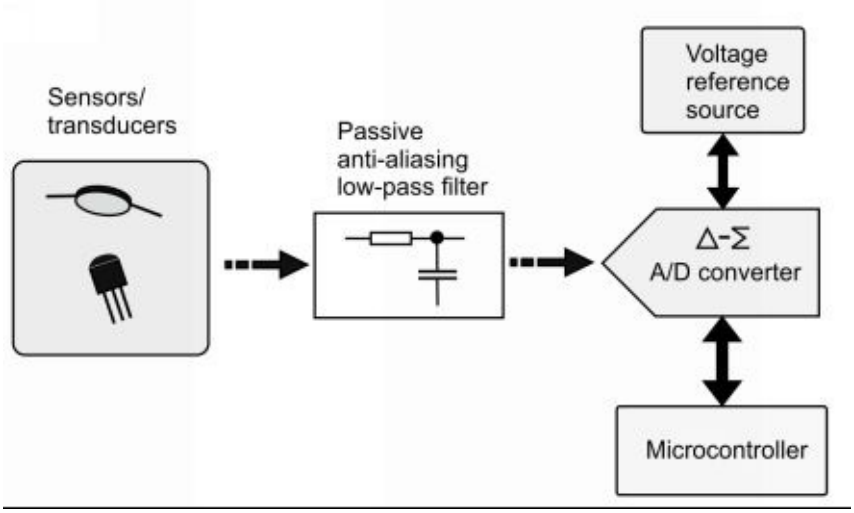


Fig.52

Circuit designers may find the simplicity of this signal chain attractive – the required external elements are the passive, anti-aliasing filter and the voltage reference.

Let's discuss the following project which illustrates using a popular low-cost Δ-Σ analog-to-digital converter MCP3551 from Microchip Corp. The MCP3551 device is 22-bit delta-sigma ADC that include fully differential analog inputs, a third-order delta-sigma modulator, a fourth-order modified SINC decimation filter, an on-chip, low-noise internal oscillator, a power supply monitoring circuit and an SPI 3-wire digital interface. These

devices can be easily used to measure low-frequency, low-level signals such as those found in pressure transducers, temperature, strain gauge, industrial control or process control applications. The power supply range for this product family is 2.7V to 5.5V.

The MCP3551/3 devices communicate with a measurement system through a simple 3-wire SPI interface. The interface controls the conversion start event, with an added feature of an auto-conversion at system power-up by tying the **CS** signal line in to logic-low. The device can communicate with bus speeds of up to 5 MHz, with 50 pF capacitive loading. The interface offers two conversion modes: ***Single Conversion*** mode for multiplexed applications and a ***Continuous Conversion*** mode for multiple conversions in series.

Every conversion is independent of each other. That is, all internal registers are flushed between conversions. When the device is not converting, it automatically goes into ***Shutdown*** mode and, while in this mode, consumes less than 1 µA. Before using this type of ADC (as well as other Δ-Σ device) it is worth learning appropriate datasheets, so that to implement the proper PCB layout and grounding.

In this demo project, the Δ-Σ converter MCP3551 is put in ***Single Conversion*** mode. The measurement system will read a positive analog input voltage from the wiper of a potentiometer and output the result in the terminal window.

The circuit diagram of the project is given below (**Fig.53**).

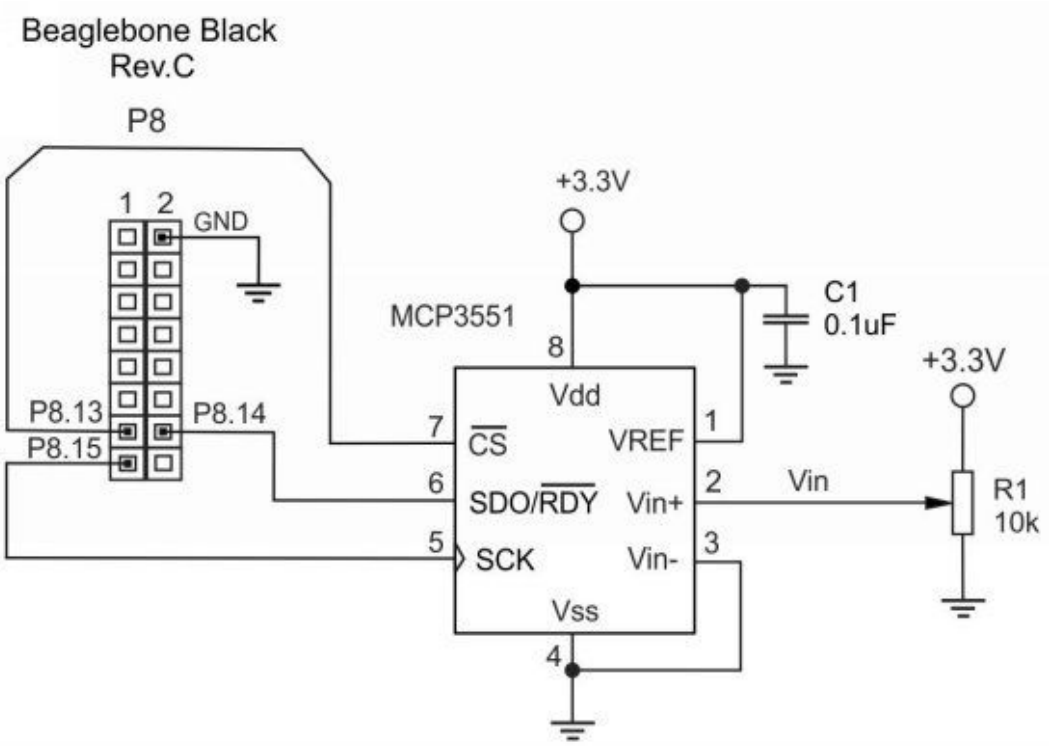


Fig.53

In this circuit, the analog input voltage is fed to the input **VIN+** (pin 2) whereas **VIN-** (pin 3) is wired to the common wire (“ground”) of the circuit. The connections between the BeagleBone board and MCP3551 Δ-Σ ADC are summarized in the table below.

BeagleBone Black P8 pin	**MCP3551 pin**
P8.15	5 (SCK)
P8.14	6 (SDO/RDY)
P8.13	7 (CS)

The reference voltage to the converter (pin 1 of MCP3551) is taken from the +3.3V DC voltage source, although for high-precision measurements you have to provide the high-stable reference taken from some voltage reference IC. The common wire of the circuit should be tied to one of the “ground” pins on the BeagleBone Black board. The bypass capacitor C1 should be tied as close as possible to pin 8 of the MCP3551 chip.

The diagram below (**Fig.54**) illustrates operating MCP3551 in ***Single Conversion*** mode.

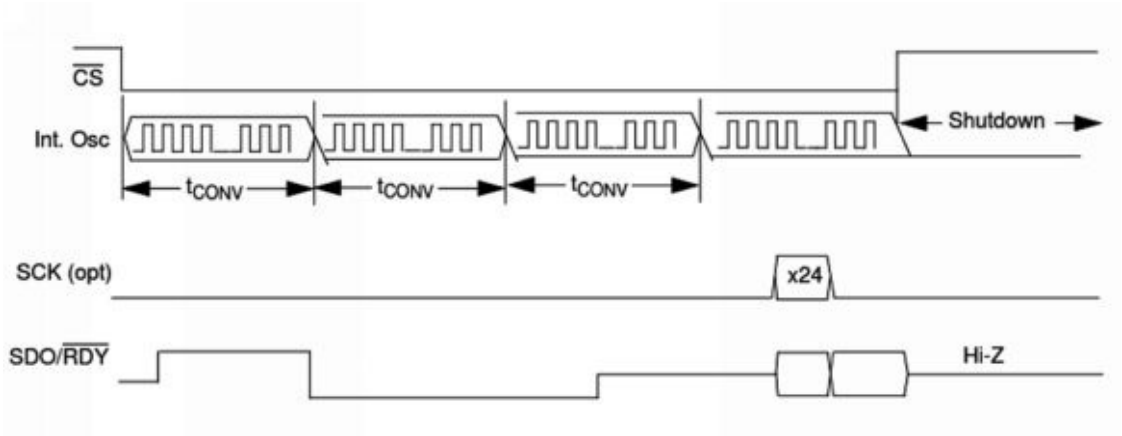


Fig.54

Serial communication between a control system and a MCP3551 device is achieved by using **CS**, **SCK** and **SDO/RDY** signals. The **CS** signal controls the conversion start. There are 24 bits in the data word: 22 bits of conversion data and two overflow bits. The conversion process takes place via the internal oscillator and the status of this conversion

must be detected by software. The status of the internal conversion is reflected by the **SDO/RDY** signal and is available with **CS** low.

A high state (log. ″ 1 ″) on **SDO/RDY**means the device is busy converting, while a low state (log. ″ 0 ″) means the conversion is finished. Line **SDO/RDY** remains in a high-impedance state when **CS** is held high. Data is ready for transfer when **SCK** goes low; each data bit is latched into a microcontroller when the **SCK** signal goes high. **CS** must be brought low before clocking out the data using **SCK** and **SDO/RDY**.

The SPI-compatible interface can operate either in (0, 0) or (1, 1) mode. In the SPI mode (0, 0) data is read using 25 clocks or four byte transfers. Note that the data ready bit (**SDO/RDY**) is included in the transfer as the first bit in this mode. When the SPI mode (1, 1) is selected, data is read using only 24 clocks or three byte transfers. The data ready bit must be read by testing the **SDO/RDY** line prior to a falling edge of the clock.

In our project we will put the SPI interface in mode (1, 1) whose timing diagram is shown in **Fig.55**.

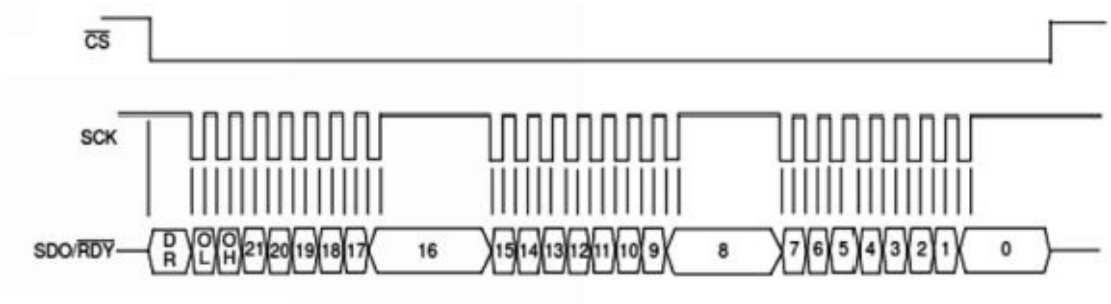


Fig.55

Bit 22 is Overflow High (OH). When $\mathbf{V_{IN}} > \mathbf{V_{REF}}$- 1 LSB, OH toggles to logic ″ 1 ″ , detecting an overflow high in the analog input voltage. Bit 23 is Overflow Low (OL). When $\mathbf{V_{IN}} < \mathbf{-V_{REF}}$, OL toggles to logic ″ 1 ″ , detecting an overflow low in the analog input voltage. The state OH = OL = ″ 1 ″ is not defined and should be considered as an interrupt for the SPI interface meaning erroneous communication.

Bit 21 to bit 0 represent the output code in 22-bit binary two’s complement. Bit 21 is the sign bit and is logic ″ 0 ″ when the differential analog input is positive and logic ″ 1 ″ when the differential analog input is negative. From Bit 20 to bit 0, the output code is given MSB first (MSB is bit 20 and LSB is Bit 0). When the analog input value is comprised between –V and V– 1 LSB, the two overflow bits are set to logic ″ 0 ″ .

Some words about checking **RDY** flag in ***Single Conversion*** mode. At every falling edge of **CS** during the internal conversion, the state of the internal conversion is latched on the **SDO/RDY** pin to give ready or busy information. A high state means the device is currently performing an internal conversion and data cannot be clocked out. A low state means the device has finished its conversion and the data is ready for retrieval on the falling edge of **SCK**. This operation is illustrated in **Fig.56**.

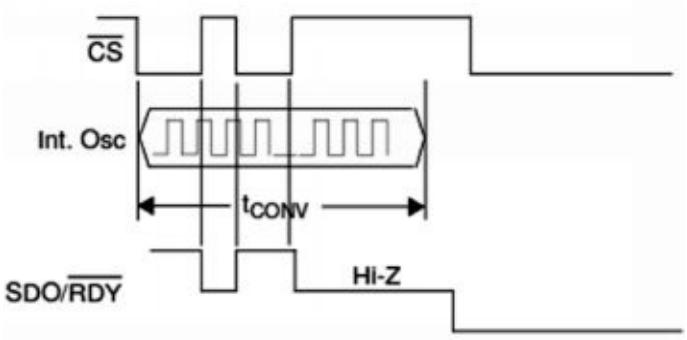


Fig.56

Note that the Ready state is latched on each falling edge of **CS** and will not dynamically update if **CS** is held low. **CS** must be toggled high then low to fix the **Ready** state.

The Python source code for driving this A/D converter is given in **Listing 23**.

Listing 23.

```
import Adafruit_BBIO.GPIO as GPIO
import time

CS = “P8_13”
SDO = “P8_14”
SCK = “P8_15”

Vref = 3.3   # Vref = 3.3 V


GPIO.setup(CS, GPIO.OUT)
GPIO.setup(SDO, GPIO.IN)
```

```
GPIO.setup(SCK, GPIO.OUT)

while True:
    binData = 0
    GPIO.output(SCK, GPIO.HIGH)
    bitDOUT = 1
    while (bitDOUT != 0x0):
    GPIO.output(CS, GPIO.HIGH)
    GPIO.output(CS, GPIO.LOW)
    bitDOUT = GPIO.input(SDO)
    i1 = 23

    while (i1 >= 0):
    GPIO.output(SCK, GPIO.LOW)
    bitDOUT = GPIO.input(SDO)
    GPIO.output(SCK, GPIO.HIGH)
    bitDOUT = bitDOUT << i1
    binData |= bitDOUT
    i1 -= 1

    GPIO.output(CS, GPIO.HIGH)
    binData &= 0x3FFFFF
    inpV = Vref * binData / 2048
    inpV = inpV / 1024
    print('MCP3551 Input Voltage:  ' + str(round(inpV, 3)) + ' V')
    time.sleep(5)
```

In this source code, the constants **SCK**, **SDO** and **CS** are associated with signals **SCK**, **SDO/RDY** and **CS** respectively. The **Vref** constant determines the reference voltage applied to pin 1 of the MCP3551 chip. In our experiment, the reference was taken from the +3.3V DC voltage source which provided the measured voltage 3.29V. To get the higher precision you should apply some stand-alone voltage reference chip.

For example, you can take an AD680 chip providing 2.5V on its output, in this case pin 1 of MCP3551 (**VREF**) should be wired to the AD680 output and the **Vref** constant should be assigned the value of 2.5. You can also take a TLV431 precision voltage reference whose output voltage can be adjusted in a wide range using a couple of resistors.

The analog-to-digital conversion consists of two phases: the internal conversion (data sampling) and data transfer. The internal conversion takes some time, so the program should check if sampling is complete before data reading. The following fragment of code accomplishes this task:

```
binData = 0
GPIO.output(SCK, GPIO.HIGH)
bitDOUT = 1
while (bitDOUT != 0x0):
    GPIO.output(CS, GPIO.HIGH)
    GPIO.output(CS, GPIO.LOW)
    bitDOUT = GPIO.input(SDO)
```

The above statements initiate the internal A/D conversion and check if it is complete by reading the **SDO/RDY** line in the **while()** loop.

When the **bitDOUT** variable is assigned 0x0, this indicates that the internal conversion has been completed and the binary data can be transferred to the BeagleBone Black.

The data transfer is performed within the **while (i1 >= 0)** loop. We need to read 24 bits, so the loop variable **i1** runs from 23 back to 0. In each iteration, the current data bit (saved in the **bitDOUT** variable) is brought to the **SDO/RDY** line on the falling edge of **SCK** and is latched into the BeagleBone board on the rising edge of **SCK**. Then the bit just taken is shifted to the proper position in the **binData** variable. The following sequence accomplishes those operations:

```
GPIO.output(SCK, GPIO.LOW)
bitDOUT = GPIO.input(SDO)
GPIO.output(SCK, GPIO.HIGH)
```

```
bitDOUT = bitDOUT << i1
binData |= bitDOUT
```

When the **while()** loop exits, the **binData** variable will hold the 24-bit value where bits 22-23 will be overflow bits and bit 21 will be a sign bit. In our experiment, we apply the positive analog voltage between 0 and 3.3V to pin 2 ($\mathbf{V_{IN+}}$), so bits 21-23 will be assigned 0 (in other cases they may take other values).

To complete the single conversion the **CS** signal line is pulled high by the following statement:

```
GPIO.output(CS, GPIO.HIGH)
```

The statement

```
binData &= 0x3FFFFF
```

leaves only 21 data bits (0 through 20) by clearing bits 21–23.

The final result taken in the float format goes to the **inpV** variable:

```
inpV = Vref * binData / 2048
inpV = inpV / 1024
```

Once we get 21–bit binary code, the LSB will be calculated as follows:

$$LSB = Vref / 2^{21} = Vref / (2^{11} \times 2^{10}) = Vref / (2048 \times 1024)$$

The running application provides the following output:

```
$ sudo python MCP3551_SigmaDelta.py
MCP3551 Input Voltage:  0.335 V
```

MCP3551 Input Voltage: 0.336 V

MCP3551 Input Voltage: 0.241 V

MCP3551 Input Voltage: 0.241 V

MCP3551 Input Voltage: 1.284 V

MCP3551 Input Voltage: 1.284 V

MCP3551 Input Voltage: 1.662 V

MCP3551 Input Voltage: 1.662 V

MCP3551 Input Voltage: 2.051 V

MCP3551 Input Voltage: 2.051 V

Digital potentiometers in programmable systems

Digital potentiometers are very versatile devices that allow to simplify design of programmable electronic circuits by replacing ordinary variable resistors. In this section we will discuss how to operate with the popular MCP41xx devices produced from Microchip Corp. Some excerpts from the relevant datasheets will clarify the matter.

According to the datasheet, the MCP41XXX/42XXX devices are 256 position single and dual digital potentiometers that can be used in place of standard mechanical potentiometers. Digital potentiometers have resistance values of 10K, 50K and 100K. In the following projects we apply a MCP41010 device which has the resistance of 10K. Each potentiometer is made up of a variable resistor and an 8-bit (2^8 = 256 position) data register that determines the wiper position. The nominal wiper resistance equals 52 Ohm for the 10K version and 125 Ohm for the 50K and 100K versions. The resistance between the wiper and either of the resistor endpoints varies linearly according to the value stored in the data register.

The MCP41XXX series provides 256 taps and accept a power supply from 2.7 through 5.5V. In our projects we feed the MCP41010 device with +5V. This also means that LSB will be equal to $5 / 2^8 = 0.0195$ V. If, for example, application writes the binary code of 100 to the device, the digital potentiometer yields $0.0195 \times 100 = 1.95$ V on its output.

Digital potentiometer applications usually fall into two categories: those that use a ″ rheostat ″ mode and applications where a ″ potentiometer ″ mode is employed. The ″ potentiometer ″ modeis frequently called a ″ voltage divider ″ mode. In the ″ rheostat ″ mode, the potentiometer acts as a two-terminal resistive element. The unused terminal should be tied to the wiper as shown in **Fig.57**.

Note that reversing the polarity of the **A** and **B** terminals will not affect operation.

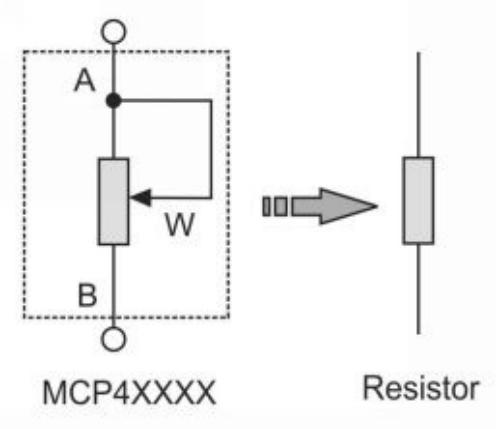


Fig.57

Using the device in this mode allows control of the total resistance between the two nodes. The total measured resistance would be the least at code 0x00, where the wiper is tied to the **B** terminal. The resistance at this code is equal to the wiper resistance, typically 52 Ohm for the 10k MCP4X010 devices, 125 Ohm for the 50k (MCP4X050), and 100k (MCP4X100) devices. For the 10k device, the LSB size would be 39.0625Ω (assuming 10k total resistance). The resistance would then increase with this LSB size until the total measured resistance at code 0xFFh would be 9985.94Ω. The wiper will never be directly connected to the **A** terminal of the resistor stack.

In the 0x00 state, the total resistance is the wiper resistance. To avoid damage of the internal wiper circuitry in this configuration, care should be taken to ensure the current flow never exceeds 1 mA.

In the ″ potentiometer ″ mode, all three terminals of the device are tied to different nodes in the circuit. This allows the potentiometer to output a voltage proportional to the input voltage (**Fig.58**). The potentiometer is used to provide a variable voltage V_2 by adjusting the wiper position between two endpoints **A** and **B**. The **A** endpoint is wired to the voltage source $\mathbf{V_1}$ and the **B**point is tied to the common wire of a circuit (″ ground ″). Note that reversing the polarity of the **A** and **B** terminals will not affect operation.

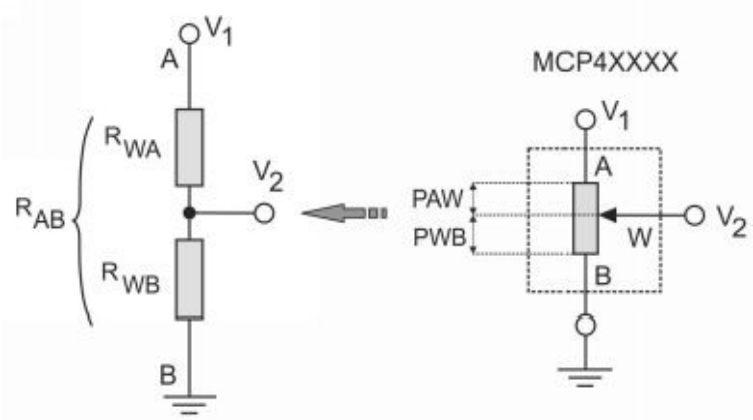


Fig.58

For the MCP41XXX devices the output analog voltage **V2** appears on the **PW** line. In the case of MCP41010 this will be pin 6 (“wiper”). Note that the digital potentiometers provide the relatively high output impedance, so an additional buffer should be inserted between the wiper and external circuitry. The buffer is also needed if an external load draws the higher current. The simple buffer may be implemented as a voltage follower built around an op-amp.

As you can see, the voltage divider is composed of the $\mathbf{R_{WA}}$ and $\mathbf{R_{WB}}$ resistors. The total resistance between **A** and **B** endpoints is determined by the parameters of a particular device. MCP41010, for instance, has the full resistance equal to 10K. The resistance of each “resistor” of this voltage divider may be calculated as follows:

$$R_{WA}(D_n) = (R_{AB}) \times (256 - D_n) / 256 + R_W$$

$$R_{WB}(D_n) = (R_{AB}) \times (D_n) / 256 + R_W,$$

where $\mathbf{R_{WA}}$ is a resistance between the terminal **PA** and the wiper (see **Fig.57**), $\mathbf{R_{WB}}$ is a resistance between the terminal **PB** and the wiper, $\mathbf{R_W}$ is a wiper resistance, $\mathbf{R_{AB}}$ is the overall resistance of the particular device and $\mathbf{D_n}$ is a 8–bit binary code that determines the output voltage of the potentiometer. In our projects we will ignore the wiper resistance because it is very low, though for very precision circuits this value should be considered. When the device is powered on, the data registers will be set to mid-scale (0x80).

Take a look at how to program the MCP41XXX chips. In order to configure $\mathbf{R_{WA}}$ and $\mathbf{R_{WB}}$ resistors we should write the predetermined 16-bit value in the data register of the digital potentiometer. The following timing diagram illustrates the write operation (**Fig.59**).

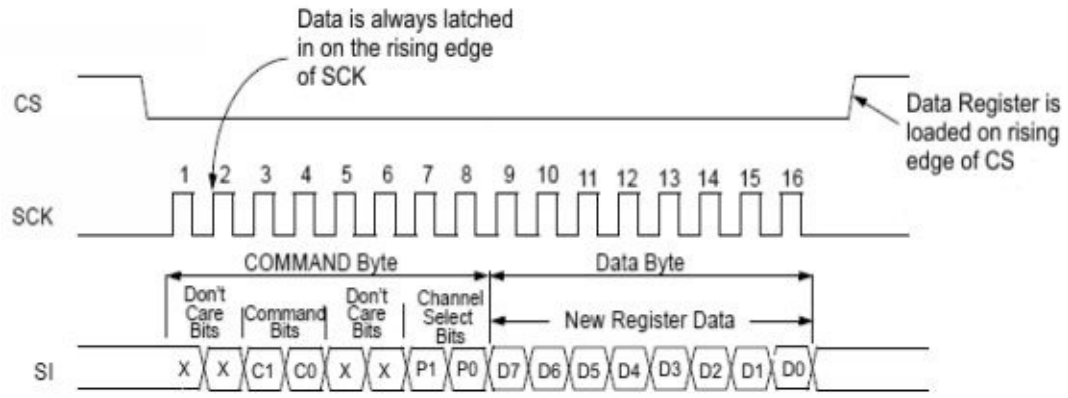


Fig.59

Writing the bit stream into the digital potentiometer is accomplished via the SPI-compatible interface. As you can see, the 16-bit binary code consists of *command* and *data* bytes. As a command byte value we will take 0x11 – this value will be used in the following projects. The data byte ($\mathbf{D_n}$) will determine the output voltage. For more detail concerning bit assignments you can explore the datasheet on the device being used.

The write operation begins when the **CS** signal goes high then low. Each data bit of the bit stream must be brought onto the **SI** line while the clock signal **SCK** is kept low. The data bit is written to the data register on the rising edge of **SCK**. After all 16 bits have been passed into the device data register, the **CS** signal goes high thus completing the operation.

The following project shows how to build the simple digitally programmed DC voltage source based upon the MCP41010 device. The circuit diagram of the project is shown in **Fig.60**.

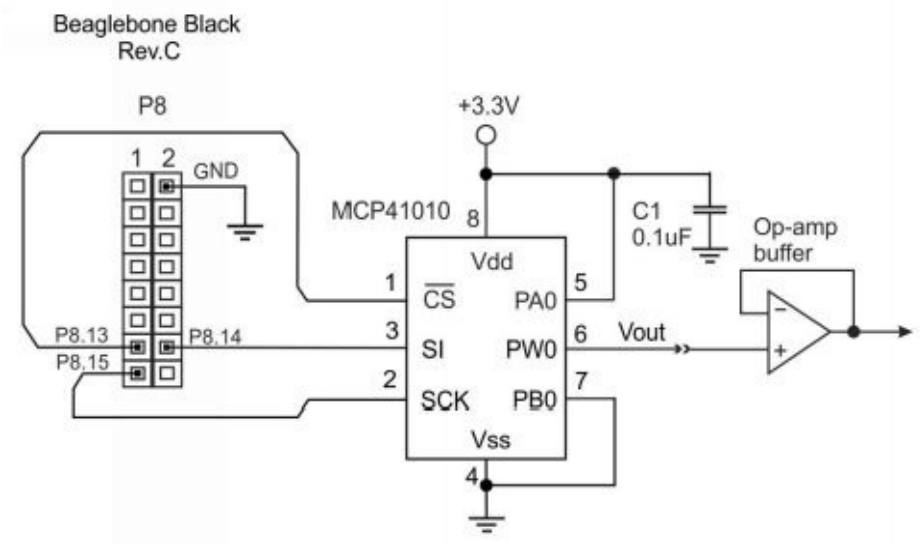


Fig.60

Here the digital potentiometer MCP41010 operates in ″ potentiometer ″ mode. The DC output voltage $\mathbf{V_{out}}$ can be taken from pin 6 of the device. MCP41010 is controlled via the SPI-compatible interface (**SCK**, **SI** and **CS** lines) from the GPIO pins of the BeagleBone Black board. The optional buffer A1 may be put between $\mathbf{V_{out}}$ and an external load to match impedances and provide the desired load capability.

The supply voltage to the MCP41010 device may be taken from a stand-alone +3.3V DC power supply. One of the “ground” pins of BeagleBone must be wired to the common wire of the circuit.

The pin connections of the above circuit are summarized in the following table.

BeagleBone Black P8 pin	**MCP41010 pin**
P8.15	2 (SCK)
P8.14	3 (SI)

P8.13	1 (CS)

The Python source code for configuring the MCP41010 device is given in **Listing 24**.

Listing 24.

```
import Adafruit_BBIO.GPIO as GPIO

SCK = “P8_15”
CS = “P8_13”
SI = “P8_14”

Vref = 3.3 # voltage reference in Volts at pin 6 of DAC

GPIO.setup(CS, GPIO.OUT)
GPIO.setup(SI, GPIO.OUT)
GPIO.setup(SCK, GPIO.OUT)

Vout = input(‘Enter desired Dig.Pot output, V:’)
fDP = float(Vout)
voutData = int((fDP / Vref) * 256)

cmd = 0x1100  # the command word
tmp = 0x0

outCode = cmd + voutData      # the data word to write

GPIO.output(CS, GPIO.HIGH)   #CS is brough high
GPIO.output(CS, GPIO.LOW)    #CS goes low thus starting conversion
```

```
i1 = 0
while (i1 < 16):
    GPIO.output(SCK, GPIO.LOW) # SCK goes low
    tmp = outCode & 0x8000
    GPIO.output(SI, GPIO.LOW)
    if(tmp != 0x0):
    GPIO.output(SI, GPIO.HIGH)
    GPIO.output(SCK, GPIO.HIGH)
    outCode = outCode << 1
    i1 += 1
GPIO.output(CS, GPIO.HIGH)
```


Here the high byte of the **outCode** variable holds the command (0x11), while the low byte (0 through 255) is kept in the **voutData** variable. The variable **voutData** is assigned the binary code corresponding to the data read from a keyboard.

Writing 16-bit value to the device is implemented within the **while()** loop. The data transfer begins when the **CS** signal changes from high to low. After LSB has been written, the program code raises the **CS** line to complete the write operation.

The measured reference voltage is 3.27V and applied to pin 5 of MCP41010. The output voltage $\mathbf{V_{out}}$ appears on the wiper **PW0**. Note that $\mathbf{V_{out}}$ depends on the voltage applied to the endpoints of the potentiometer (3.27V, in our case).

We can replace the MCP41010 part by the similar device, for example, MCP41050 with the total resistance of 50k or MCP41100 with the resistance of 100k. This application can easily be improved when dual channel digital potentiometers MCP42XXX are required, though such evolution will require modifications in both hardware and software.

The wide-band oscillator LTC1799 using a digital potentiometer

A very simple digitally controlled oscillator can be developed using a popular LTC1799 device from Linear Technology Corp. LTC1799 is easy to use precision oscillator designed for high accuracy operation (≤1.5% frequency error) without the need for external trim components. The LTC1799 device operates with a single 2.7V to 5.5V power supply and provides a rail-to-rail, 50% duty cycle square wave output. The CMOS output driver ensures fast rise/fall times and rail-to-rail switching.

The oscillator circuit with LTC1799 is shown in **Fig.61.**

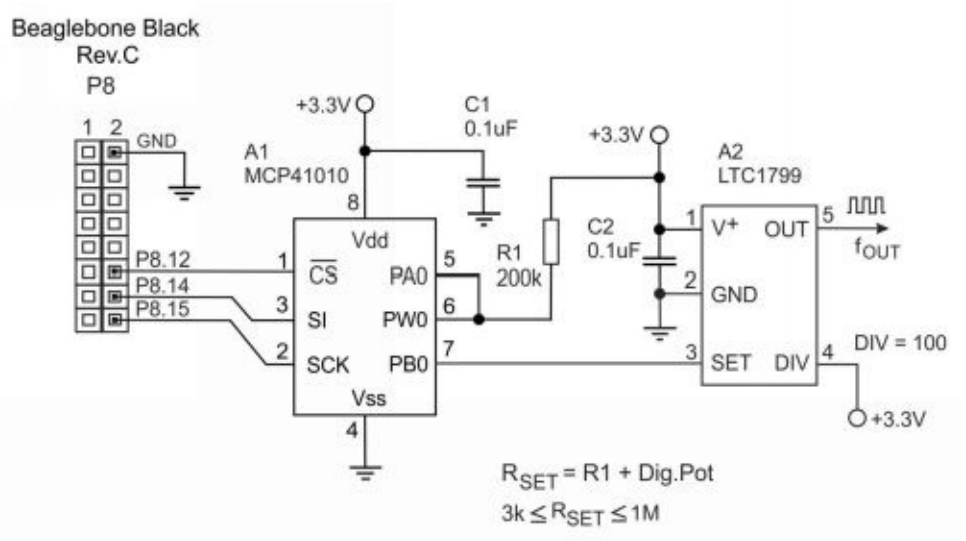


Fig.61

The oscillator frequency is programmed by a single external resistor (**R_{SET}**) using the formula shown in **Fig.62**.

$$f_{OSC} = 10MHz \times \left(\frac{10k}{N \times R_{SET}}\right)$$

$$N = \begin{cases} 100, & \text{DIV Pin} = V^+ \\ 10, & \text{DIV Pin} = \text{Open} \\ 1 & \text{DIV Pin} = \text{GND} \end{cases}$$

Fig.62

The frequency-setting resistor can vary from 3K to 1M to select a master oscillator frequency between 1KHz and 33MHz. In this circuit, the **RSET** consists of the R1 resistor (200k) and the digital potentiometer **PA0-PW0-PB0** (10k) connected in series. When the resistance of the **PA0-PW0-PB0** reaches 0k, the output frequency will achieve 5000 Hz; at resistance of 10k, the output frequency decreases to 4750 Hz. Thus we can change the output frequency in the range from 4750 to 5000 Hz by adjusting the value of the digital potentiometer.

The three-state **DIV** input determines whether the master clock is divided by 1, 10 or 100 before appearing on the LTC1799 output. Therefore, **DIV** determines three frequency ranges spanning 1KHz to 33MHz. In our circuit, the output frequency is divided by 100, since the **DIV** pin is connected to +3.3V. LTC1799 may be replaced by the LTC6900 oscillator that is a lower power version of the LTC1799.

Note that the pin configuration of the LTC1799 chip in **Fig.60** relates to the 5-Lead Plastic TSOT-23 package.

The digitally programmed wide-band oscillator with LTC6903

Most embedded systems use various periodical digital signals (″ pulse trains ″). BeagleBone Black can be programmed to generate pulse trains using GPIO pins, although the frequency of such signals will reach only a few KHz. In order to build a clock source which frequency can span in a wide band, we can employ some single-chip oscillator driven by either the SPI or I^2C interface.

This project illustrates the design of a digitally programmed wide–band oscillator using a popular chip LTC6903 from Linear Technology. This chip provides a TTL-compatible high-stable and high-accuracy signal in the range from 1 KHz through 68 MHz. LTC6903 is driven through the SPI-compatible interface, so we can connect the device to the GPIO pins of the BeagleBone Black.

The brief description of LTC6903 excerpted from its datasheet is given below.

The LTC6903 device is a low-power, self contained digital frequency source providing a precision frequency from 1 KHz to 68 MHz, setable through a serial port. LTC6903 requires no external components other than a power supply bypass capacitor, and it operates over a single wide supply range of 2.7V to 5.5V. In our project the supply voltage will be 3.3V.

As the datasheet says, the LTC6903 oscillator chip features a proprietary feedback loop that linearizes the relationship between the digital control setting and the output frequency, resulting in a very simple frequency setting equation:

$f = 2^{OCT} \times 2078(Hz) / (2 - DAC/1024)$ (1),

where 1 KHz < f < 68 MHz, OCT is a 4-bit integer value (0-15) represented by the serial port register bits OCT [3:0] and DAC is a 10-bit integer value (from 0 through 1023) represented by the serial port register bits DAC[9:0].

The output frequency of LTC6903 can be set programmatically via a 3-wire SPI-compatible interface controlled by the BeagleBone Black board (**Fig.63)**.

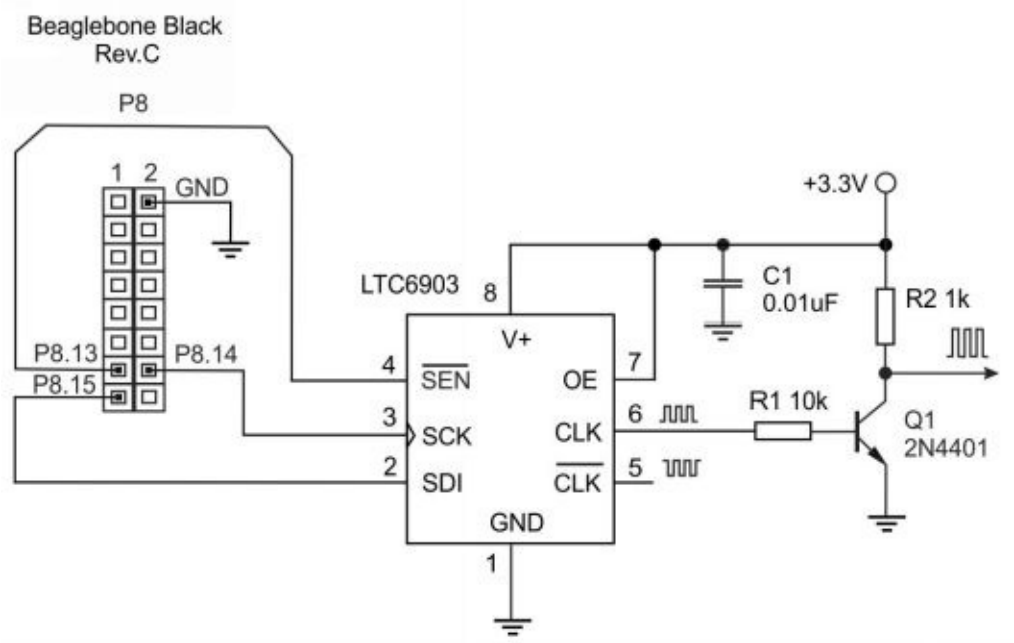


Fig.63

Here pin **P8.13** of the BeagleBone board is connected to the signal line **SEN** of the LTC6903 chip, **P8.14** goes to the **SCK** line and **P8.15** is connected to the data line **SDI**. One of the "ground" pins of the BeagleBone board must be wired to the common wire of the circuit.

The pin connections of the above circuit are summarized in the following table.

BeagleBone Black P8 pin	**LTC6903 pin**
P8.15	2 (SDI)
P8.14	3 (SCK)
P8.13	4 (SEN)

The output frequency appears on the **CLK** output (pin 6 of LTC6903). The inverse output signal can be taken from pin 5; we will not use this output, so pin 5 is left unconnected. The **OE** signal line wired to the source voltage +3.3V allows both **CLK** outputs. In order to reduce the noise and increase the accuracy of the pulse train the bypass capacitor C1 of about 0.01uF is tied close to the power pins of the LTC6903 chip. It is worth cutting off any excess of the capacitor leads as much as possible to minimize their series inductance.

Take a look at how to configure the LTC6903's output frequency.

The desired output frequency of LTC6903 can be set by applying formula (1). To

configure the output frequency the following steps are required:

1. Selection of the appropriate value of OCT from the table shown below:

Output Frequency Range vs. OCT Setting (Frequency Resolution 0.001 * f)

f ≥	**f <**	**OCT**
34.05 MHz	68.03 MHz	15
17.02 MHz	34.01 MHz	14
8.511 MHz	17.01 MHz	13
4.256 MHz	8.503 MHz	12
2.128 MHz	4.252 MHz	11
1.064 MHz	2.126 MHz	10
532 KHz	1063 KHz	9
266 KHz	531.4 KHz	8
133 KHz	265.7 KHz	7
66.5 KHz	132.9 KHz	6
33.25 KHz	66.43 KHz	5
16.62 KHz	33.22 KHz	4
8.312 KHz	16.61 KHz	3
4.156 KHz	8.304 KHz	2
2.078 KHz	4.152 KHz	1
1.039 KHz	z	0

2. Choosing the variable DAC through the following formula, and rounding DAC to nearest integer:

DAC = 2048 – 2078(Hz) × 2^(10–OCT) / 1 (2)

3. After OCT and DAC have been determined, we should make up the 16-bit binary code to be written to the serial register of the LTC6903 chip (**Fig.64**):

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
OCT3	OCT2	OCT1	OCT0	DAC9	DAC8	DAC7	DAC6	DAC5	DAC4	DAC3	DAC2	DAC1	DAC0	0	0

Fig.64

In this particular example, the **D1** – **D0** fields are assigned 0. Once the 16-bit code has been determined, we can write it into the LTC6903 device using the timing diagram shown in **Fig.65**.

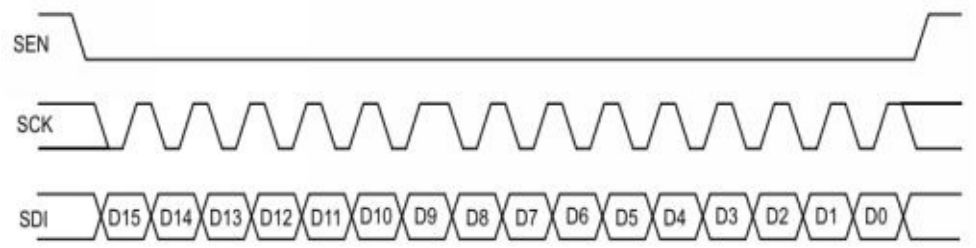


Fig.65

As it is seen, the **SEN** signal frames the write operation which transfers the 16-bit data stream into the serial register of LTC6903. When **SEN** goes low, the data transfer is enabled. The program code puts each bit on the **SDI** line (MSB goes first) and latches this bit into the device on the rising edge of the **SCK** pulse. After the last bit (D0) has been written, **SEN** goes high.

The Python source code driving the LTC6903-based oscillator is shown in **Listing 25**.

Listing 25.

```
import Adafruit_BBIO.GPIO as GPIO

SEN = “P8_13”
```

```
SCK = “P8_14”
SDI = “P8_15”

GPIO.setup(SEN, GPIO.OUT)
GPIO.setup(SDI, GPIO.OUT)
GPIO.setup(SCK, GPIO.OUT)



#freq = 0x844   #F = 1400 Hz

#freq = 0x1774  #F = 2710 Hz
#freq = 0x1C54  #F = 3380 Hz
#freq = 0x2274  #F = 4500 Hz
#freq = 0x262C  #F = 5150 Hz

#freq = 0x2E1C   #F = 7432 Hz
freq = 0x4250  #F = 17920 Hz
#freq = 0x4C6C  #F = 27168 Hz
#freq = 0x5048  #F = 33540 Hz

GPIO.output(SEN, GPIO.HIGH)
GPIO.output(SEN, GPIO.LOW)

i1 = 0
while (i1 < 16):
    GPIO.output(SCK, GPIO.LOW)
    tmp = freq & 0x8000
    GPIO.output(SDI, GPIO.LOW)
    if (tmp != 0x0):
    GPIO.output(SDI, GPIO.HIGH)
    GPIO.output(SCK, GPIO.HIGH)
```

```
    freq = freq << 1
    i1 += 1
```



```
GPIO.output(SEN, GPIO.HIGH)
```

The program controls the data transfer through three GPIO pins (**P8.13 – P8.15)** configured as outputs by the following sequence:

```
GPIO.setup(SEN, GPIO.OUT)   # set P8.13 as output
GPIO.setup(SCK, GPIO.OUT)   # set P8.14 as output
GPIO.setup(SDI, GPIO.OUT)    # set P8.15 as output
```

The **SCK** pin produces the clock signal **SCK**, **SEN** provides the framing signal on the **SEN** line and **SDI** keeps the bit shifted onto the **SDI** line.

The **freq** variable keeps the value of the desired output frequency. In our case, the output frequency is set to 17920 Hz, so **freq** is assigned the value **0x4250**. The **tmp** variable holds the current MSB being taken out of the **freq** variable in each iteration.

Before writing the data word into the LTC6903 registers, the **SEN** signal is brought high, then low. The low level of **SEN** enables the data transfer. The following two statements accomplish that:

```
GPIO.output(SEN, GPIO.HIGH)
GPIO.output(SEN, GPIO.LOW)
```

Since the data word contains 16 bits (D15 through D0), the

while (i1 < 16)

loop is entered. The loop will run until all 16 iterations have passed. In each iteration, the current MSB is driven on the **SDI** line while the **SCK** signal is kept low. This data bit is then written to the LTC6903 device by bringing the **SCK** signal high. The single iteration is executed by the following sequence:

```
GPIO.output(SCK, GPIO.LOW)
```

```
tmp = freq & 0x8000
GPIO.output(SDI, GPIO.LOW)
if (tmp != 0x0):
    GPIO.output(SDI, GPIO.HIGH)
GPIO.output(SCK, GPIO.HIGH)
freq = freq << 1
i1 += 1
```

The

```
freq = freq << 1
```

statement in this sequence shifts all bits in **freq** to the left by 1 in preparation for the next iteration. After all 16 data bits have been processed, the loop exits. The framing signal **SEN** that follows goes high thus disabling the write operation:

```
GPIO.output(SEN, GPIO.HIGH)
```

To set the output frequency other than that in this project, you should recalculate the value of the **freq** variable.

Using a digital-to-analog converter as a precision DC voltage source

Digitally controlled high-stable analog signals may be produced by stand-alone digital-to-analog converters (D/A converter, DAC). A common 12-bit DAC powered from +3.3V voltage source can provide the resolution of $3.3/2^{12} = 3.3/4096 = 0.0008$ V that is acceptable for many applications where high precision is needed.

The following project illustrates interfacing and programming the popular low-cost 12-bit DAC MCP4921 from Microchip Corp. The circuit diagram of the project is shown in **Fig.66.**

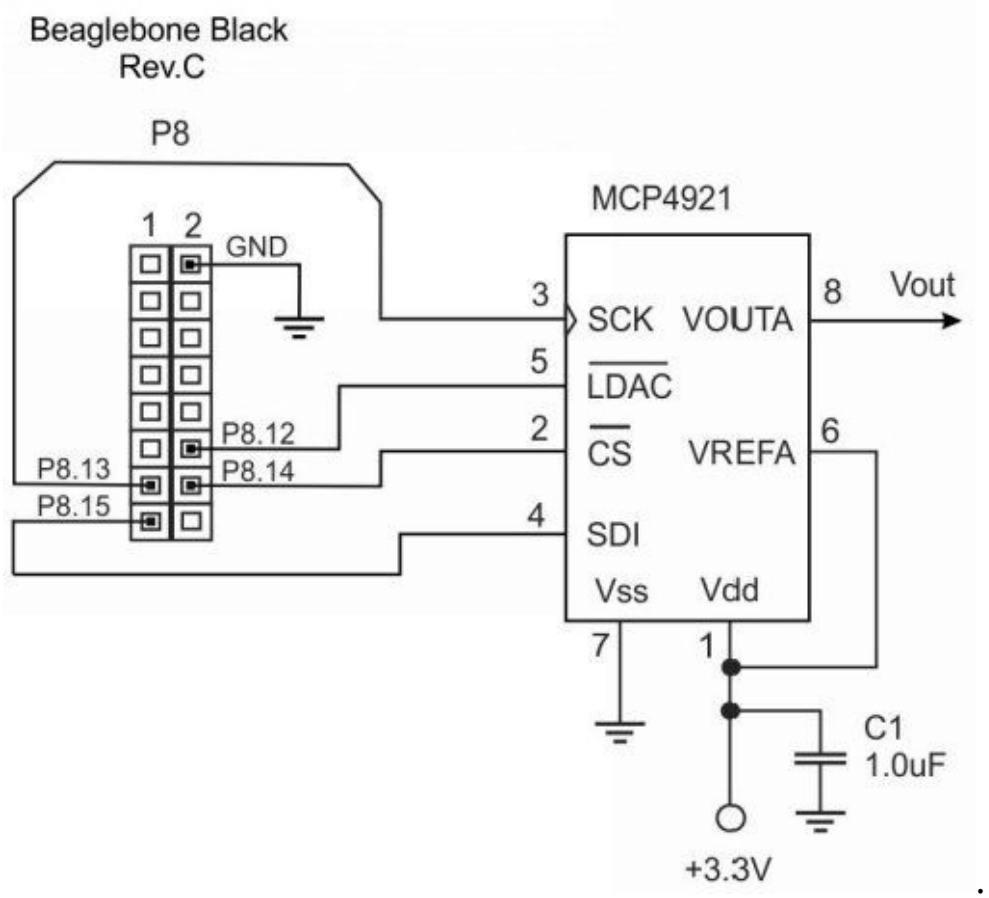


Fig.66

The connections between the BeagleBone Black and the MCP4921 DAC are summarized in the table below:

BeagleBone Black P8 pin	**MCP4921 pin**

P8.13	3 (SCK)
P8.14	2 (CS)
P8.15	4 (SDI)
P8.12	5 (LDAC)

One of the “ground” pins of the BeagleBone board must be tied to the common wire of the circuit.

A timing diagram of the MCP4921 device shown in **Fig.67** determines the relationships between interface signals while performing the D/A conversion. Note that most DACs use similar timing diagrams, so understanding MCP4921 can help when dealing with other DAC chips.

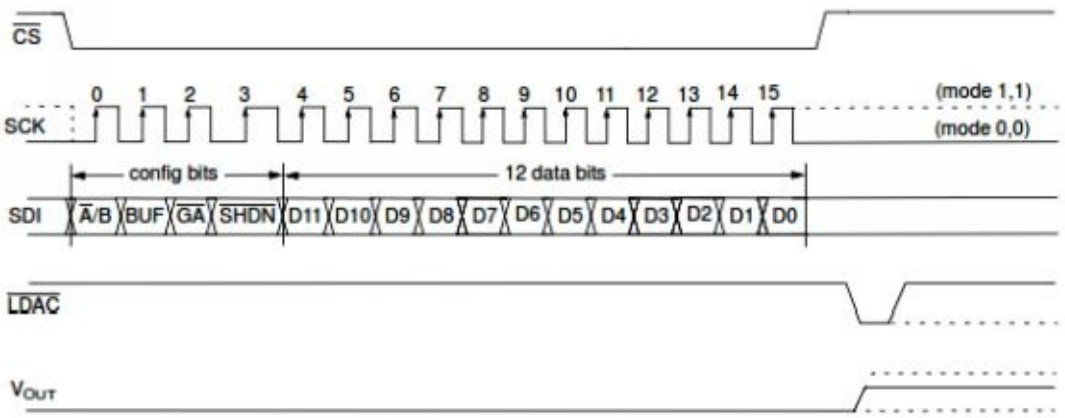


Fig.67

The MCP4921 DAC is designed to operate through a serial SPI-compatible interface, so it is easily to drive this device from the BeagleBone Black board. Commands and data are sent to DAC via the **SDI** pin, with data being clocked in on the rising edge of **SCK**. The communications are unidirectional, so data cannot be read out of the MCP492X devices. The **CS** pin must be held low for the duration of a write command.

The write command is composed of 16 bits and is used to configure the DAC’s control and data latches. Writing is initiated by pulling the **CS** pin low, followed by clocking four configuration bits and 12 data bits into the **SDI** pin on the rising edge of **SCK**. After processing all bits the **CS** signal must be brought high, thus causing the data to be latched into the selected DAC’s input registers. The MCP4921 chip includes a double-buffered latch structure to allow DAC output to be synchronized with the **LDAC** pin, if necessary.

When the **LDAC** signal goes low, the value held in the DAC input register is transferred into the DAC output register. Since all write operations to the MCP492X are 16-bit wide, any clocks past 16 will be ignored. The most significant four bits (15 – 12) are configuration bits, while the remaining 12 bits are data bits. When **CS** goes high, no data can be transferred into the device. **Note** that if the **CS** signal is driven high before all 16 bits have been transferred, shifting the data into the input registers will be aborted.

Before running a write operation we need to make up a 16-bit word whose bits are described in the table below.

Bit	**Name**	**Description**
15	A/B	″ 1 ″ means the data will be written to DACB, ″ 0 ″ allows writing to DACA
14	BUF	″ 1 ″ enables the buffered mode of the input, while ″ 0 ″ puts the input to the unbuffered mode
13	GA	″ 1 ″ sets the gain of the output signal equal to 1, ″ 0 ″ allows to double the magnitude of the output signal
12	SHDN	″ 1 ″ allows all operations on DAC, ″ 0 ″ disables the output buffer although write operations are still allowed
11–0	D11–D0	Data bits which represent the 12-bit number between 0 and 4095

For the given case, we will set the configuration bits 15 – 12 to 0111 (0x7 in the hexadecimal notation).

The source code for driving the MCP4921 DAC is shown below (**Listing 26**).

Listing 26.

```
import Adafruit_BBIO.GPIO as GPIO
import math

SCK = “P8_13”   # P8.13 —-> SCK
CS = “P8_14”     # P8.14 —-> CS
SDI = “P8_15”    # P8.15 —-> SDI
LDAC = “P8_12” # P8.12 —-> LDAC

Vref = 3.3 # voltage reference in Volts at pin 6 of DAC

GPIO.setup(CS, GPIO.OUT)
GPIO.setup(SDI, GPIO.OUT)
GPIO.setup(SCK, GPIO.OUT)
GPIO.setup(LDAC, GPIO.OUT)

Vout = input(‘Enter desired DAC output, V:’)
fDAC = float(Vout)
binCode = int((fDAC / Vref) * 4096)

cmd = 0x7000         # the highest 4 bits compose the command
tmp = 0x0

fword = cmd | binCode      # making up word to write
GPIO.output(CS, GPIO.HIGH) # CS is brought high
GPIO.output(CS, GPIO.LOW)  # CS goes low thus enabling the write operation
GPIO.output(LDAC, GPIO.HIGH)  # LDAC goes high
i1 = 0

while (i1 < 16):
    GPIO.output(SCK, GPIO.LOW)   # SCK goes low
    tmp = fword & 0x8000
```

```
    GPIO.output(SDI, GPIO.LOW)
    if(tmp != 0x0):
    GPIO.output(SDI, GPIO.HIGH)
    GPIO.output(SCK, GPIO.HIGH)
    fword = fword << 1
    i1 += 1
GPIO.output(CS, GPIO.HIGH)
GPIO.output(LDAC, GPIO.LOW)
GPIO.output(LDAC, GPIO.HIGH)
```


Here the **cmd** variable is assigned the command with a code 0x7. The **binCode** variable represents the binary code for the DAC's output. The program waits until a user has entered the value of the desired DAC output voltage (the **input()** function), then the data of a float type will be assigned to the **fDAC** variable.

The binary code representing the DAC output voltage is stored in the **binCode** variable after executing the following statement:

```
binCode = int((fDAC / Vref) * 4096)
```

The full 16-bit word being written to the DAC (**fword** variable) is the combination of the values stored in the **cmd** and **binCode** variables:

```
fword = cmd | binCode
```

The **SCK**, **SDI**, **CS** and **LDAC** constants determine the pins connected to the DAC signal lines. Since we are allowed only to transfer the data to MCP4921, all pins are configured as outputs.

The data transfer requires 16 iterations running within the **while()** loop with the loop variable **i1** being incremented from 0 to 15. Before the loop is entered, the conversion is enabled by pulling the **CS** line high, then low. The **LDAC** line must be high. This is done by following sequence:

```
GPIO.output(CS, GPIO.HIGH)
```

```
GPIO.output(CS, GPIO.LOW)
GPIO.output(LDAC, GPIO.HIGH)
```

In each iteration, the current MSB of **fword** is taken and stored in the **tmp** variable:

```
tmp = fword & 0x8000;
```

Depending on the value held in the **tmp** variable, the **SDI** line is set either high or low. Shortly afterwards the bit on **SDI** is clocked out by the **SCK** signal. The following statements perform those operations:

```
GPIO.output(SDI, GPIO.LOW)
if(tmp != 0x0):
    GPIO.output(SDI, GPIO.HIGH)
GPIO.output(SCK, GPIO.HIGH)
```

The value in the **fword** variable is then shifted to the left by 1 in preparation for the next iteration:

```
fword = fword << 1;
```

When the **while()** loop exits, the **CS** signal is pulled high thus disabling the data transfer. The data obtained will be written to the output buffer of DAC by bringing the **LDAC** line low. The following sequence performs those steps:

```
GPIO.output(CS, GPIO.HIGH)
GPIO.output(LDAC, GPIO.LOW)
GPIO.output(LDAC, GPIO.HIGH)
```

To test the application we should save the source code in the MCP4921_DAC_Test.py file and launch this program. While running, the program prompts a user to enter the desired DAC output voltage:

```
$ sudo python MCP4921_DAC_Test.py
Enter DAC output voltage, V:0.55
$ sudo python MCP4921_DAC_Test.py
Enter DAC output voltage, V:2.39
$ sudo python MCP4921_DAC_Test.py
Enter DAC output voltage, V:0.16
```

The voltage level at the DAC’s output (pin 8 of MCP4921) can easily be checked using a multimeter or voltmeter.

The digital-to-analog converter in the LTC6992 PWM circuit

This section describes the use of a versatile chip LTC6992 from Linear Technology which can be applied as a " Voltage-to-Duty Cycle " converter. The brief description of LTC6992 taken from its datasheet is given below.

The LTC6992 is a silicon oscillator with an easy-to-use analog voltage-controlled pulse width modulation (PWM) capability. The LTC6992 is part of the TimerBlox family of versatile silicon timing devices. The LTC6992 operates with a single 2.7V to 5.5V power supply and provides the frequency range from 3.81 Hz to 1 MHz.

A single resistor, R_{SET}, programs the LTC6992's internal master oscillator frequency. The output frequency is determined by this master oscillator and an internal frequency divider, N_{DIV}, programmable to eight settings from 1 to 16384 (**Fig.68**).

$$f_{OUT} = \frac{1MHz}{N_{DIV}} \times \frac{50k}{R_{SET}}$$

$$N_{DIV} = 1, 4, 16 \ldots 16384$$

Fig.68

Applying a voltage between 0V and 1V on the **MOD** pin sets the duty cycle. The four versions differ in their minimum/maximum duty cycle (see the table below).

DEVICE NAME	**PWM DUTY CYCLE RANGE**
LTC6992-1	0% to 100%
LTC6992-2	5% to 95%
LTC6992-3	0% to 95%
LTC6992-4	5% to 100%

Note that a minimum duty cycle limit of 0% or maximum duty cycle limit of 100% allows oscillations to stop at the extreme duty cycle settings.

The circuit shown in **Fig.69** allows to generate the PWM signal with a variable duty cycle which can be adjusted by the voltage on the DAC MCP4921 output. The DAC, in turn, is driven by the BeagleBone Black software, so we can set the PWM duty cycle programmatically. **Note**that the pin configuration of the LTC6992 chip corresponds to the 6-Lead Plastic TSOT-23 package.

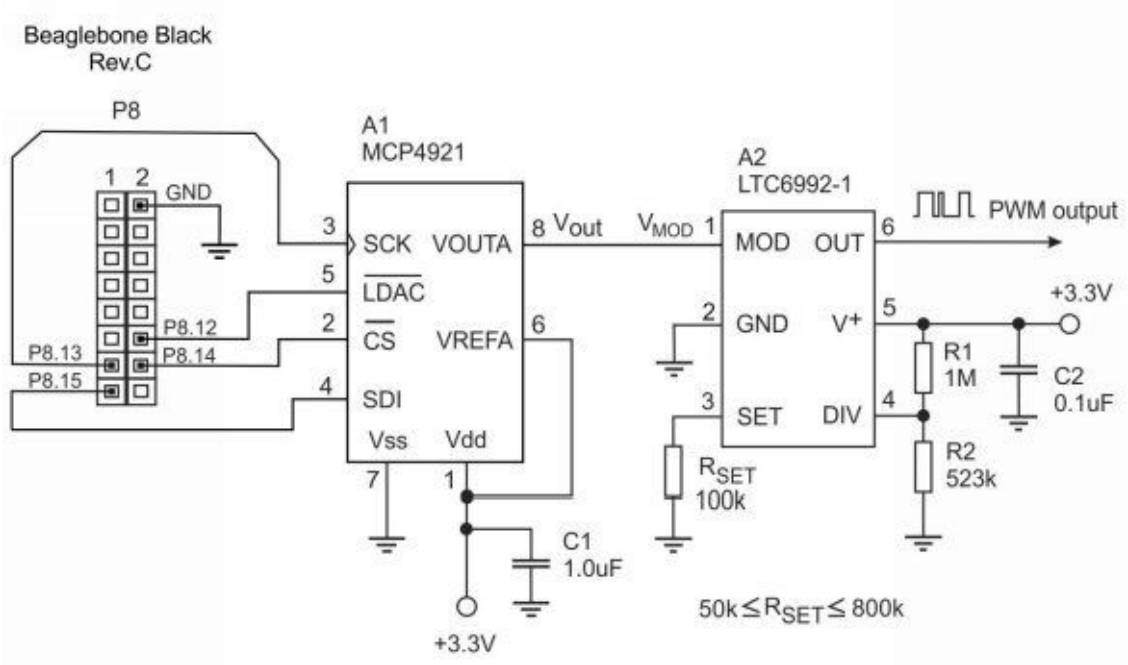


Fig.69

In this circuit, the LTC6992-1 chip provides the output pulse train on pin 6 whose duty cycle will be linearly proportional to the modulated voltage $\mathbf{V_{MOD}}$ provided by the D/A converter. The signal $\mathbf{V_{MOD}}$ arrives at the Pulse Width Modulation Input (**MOD**, pin 1 of LTC6992-1). The linear control of the duty cycle should range between approximately 100mV to 900mV on the **MOD** pin. Beyond those limits, the output will either clamp at 5% or 95%, or stop oscillating (0% or 100% duty cycle), depending on the version.

The resistor R_{SET} which determines the oscillation frequency is connected to the **SET** input (pin 3). In our case, the R_{SET} is taken equal to 100K. The **DIV** input (pin 4) is a Programmable Divider and Polarity Input.

The **DIV** pin connects to an internal, V+ referenced 4-bit A/D converter that determines the **DIVCODE** value. **DIVCODE** programs two settings on the LTC6992. First, it determines the output frequency divider setting, N_{DIV}. Second, **DIVCODE** determines the output polarity.

The input voltage to **DIV** is generated by the resistor divider R1-R2 connected between **V+** and “ground”. To ensure an accurate result a developer must use at least 1% resistors. The **DIV** pin and resistors should be shielded from the **OUT** pin or any other traces that

have fast edges.

In this circuit, the N_{DIV} is set to 1024, thus determining the output frequency (**OUT**, (pin 6 of LTC6992-1) to be 488 Hz (see formula in **Fig.68**). Other possible values of N_{DIV} can be calculated using the approach described in the device datasheet.

The relationship between the analog voltage $\mathbf{V_{MOD}}$ applied to the **MOD** input (pin 1) and the duty cycle of the output frequency is given by the following formula (**Fig.70**).

$$\text{Duty Cycle} = D = \frac{V_{MOD} - 100mV}{800mV}$$

$$V_{MOD} = (D \times 800mV) + 100mV$$

Fig.70

Knowing the value of the duty cycle D, it is easily to calculate the value of the input voltage $\mathbf{V_{MOD}}$ applied to the MOD input of LTC6992-1. The basic source code for driving LTC6992-1 PWM circuit can be the same as that shown in **Listing 26**, although a few modifications should be done.

Generating analog waveforms using the Direct Digital Synthesizer AD9850

Direct Digital Synthesis (DDS) is a method of producing an analog waveform (usually a sine wave) by generating a time-varying signal in digital form and then performing a digital-to-analog conversion. Because operations within a DDS device are primarily digital, this gives fast switching between output frequencies, fine frequency resolution, and operation over a broad spectrum of frequencies.

The ability to accurately produce and control waveforms of various frequencies and profiles has become a key requirement common to a number of industries. Many possibilities for frequency generation are open to a designer, but the DDS technique is rapidly gaining acceptance for solving frequency (or waveform) generation requirements in both communications and industrial applications because single-chip IC devices can generate programmable analog output waveforms simply and with high resolution and accuracy. By using modern DDS chips, we can obtain sine wave, triangular and rectangle signals with low-level total harmonic distortion (THD).

This section describes the programmable waveform oscillator using a popular DDS chip AD9850 which can produce the sine wave and square wave signals in a wide range of frequencies. The device output frequency is controlled through the SPI-compatible interface by the BeagleBone Black board. Before we start describing the hardware/software let’s see on how AD9850 works. The brief description that follows is taken from the datasheet on the device.

The AD9850 is a highly integrated device that uses advanced DDS technology coupled with an internal high speed, high performance D/A converter and comparator to form a complete, digitally programmable frequency synthesizer and clock generator function. When referenced to an accurate clock source, the AD9850 generates a spectrally pure, frequency/phase programmable, analog output sine wave. This sine wave can be used directly as a frequency source, or it can be converted to a square wave for agile-clock generator applications.

The AD9850’s innovative high speed DDS core provides a 32-bit frequency tuning word, which results in an output tuning resolution of 0.0291 Hz for a 125 MHz reference clock input. The AD9850’s circuit architecture allows the generation of output frequencies of up to one-half the reference clock frequency (or 62.5 MHz), and the output frequency can be digitally changed (asynchronously) at a rate of up to 23 million new frequencies per second. The device also provides five bits of digitally controlled phase modulation, which

enables phase shifting of its output in increments of 180°, 90°, 45°, 22.5°, 11.25°, and any combination thereof. The AD9850 also contains a high speed comparator that can be configured to accept the (externally) filtered output of the DAC to generate a low jitter square wave output. This facilitates the device's use as an agile clock generator function.

The frequency tuning, control, and phase modulation words are loaded into the AD9850 via a parallel byte or serial loading format. The parallel load format consists of five iterative loads of an 8-bit control word (byte). The first byte controls phase modulation, power-down enable, and loading format; Bytes 2 to 5 comprise the 32-bit frequency tuning word. Serial loading is accomplished via a 40-bit serial data stream on a single pin. The AD9850 Complete DDS uses advanced CMOS technology to provide this breakthrough level of functionality and performance on just 155 mW of power dissipation (3.3 V supply).

The following circuit diagram (**Fig.71**) illustrates connecting AD9850 to the BeagleBone Black.

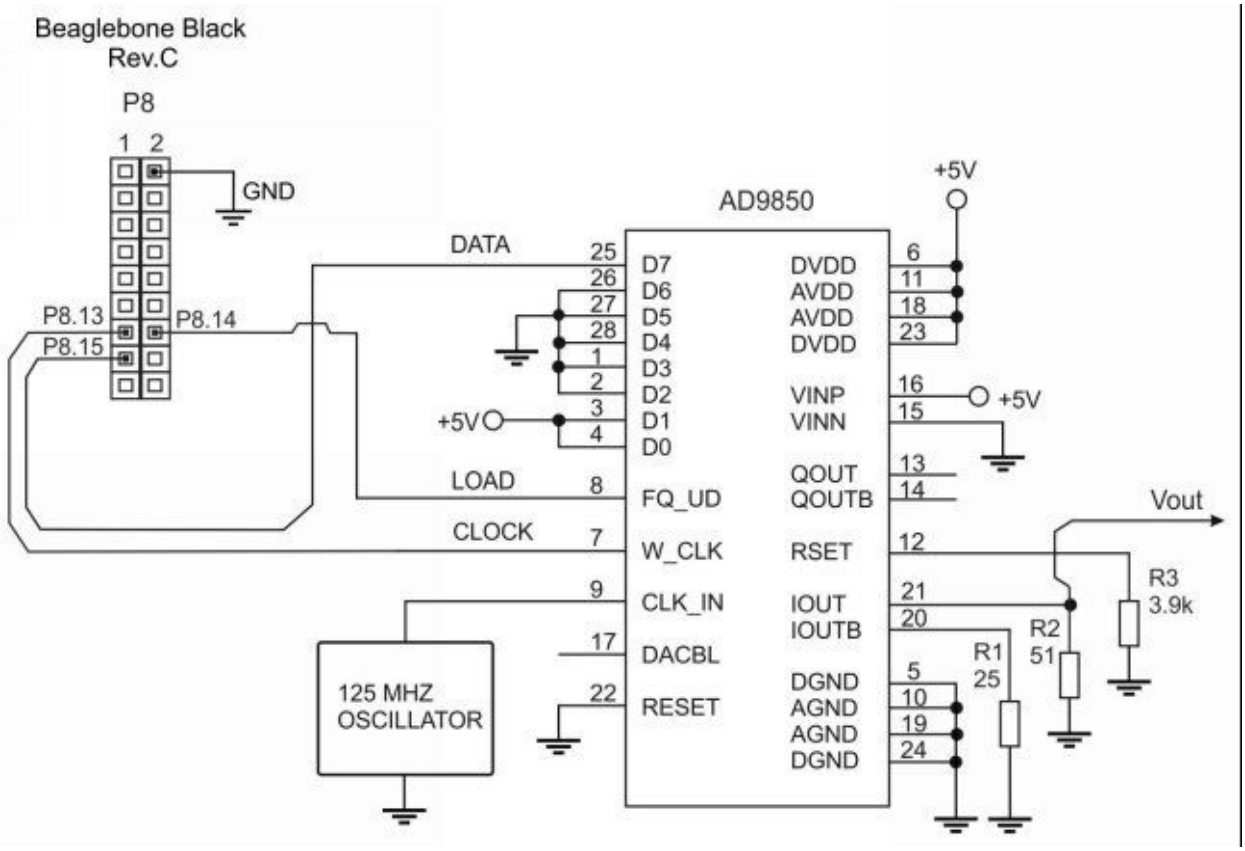


Fig.71

Assembling such a circuit can be a hard work, so it would be much better to apply one of numerous low-cost ready-to-use modules with AD9850. One of such modules taken to this project is shown in **Fig.72**.

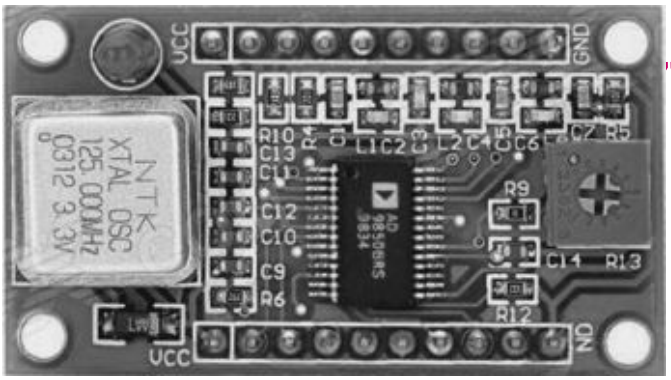


Fig.72

It is seen that the reference clock fed to the AD9850 chip is 125 MHz, therefore the output frequency can reach 62.5 MHz. As to this module, it is stated that the real output frequency can reach 40 MHz because of the low-pass filter placed at the DAC output of AD9850. However, the developers experimenting with this module will be capable to pull up the output to its maximal frequency.

The project described here uses the above module. **Fig.73** shows connections between the AD9850 module and the BeagleBone Black board.

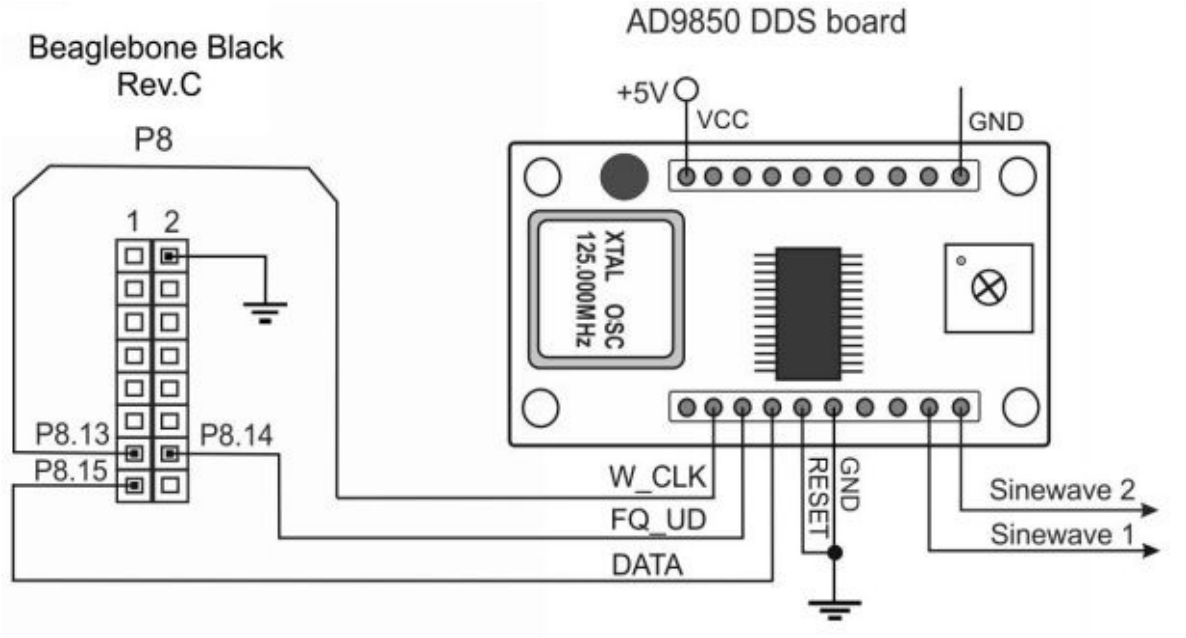


Fig.73

In this circuit, the **DATA** line (pin D7 of AD9850) is connected to pin **P8.15** of the BeagleBone. The framing signal **FQ_UD** goes to pin **P8.14** and the clock input **W_CLK** of AD9850 goes to pin **P8.13** of the BeagleBone Black.

Note that the connections in circuit shown in **Fig.73** are specific for my own AD9850 board purchased at www.dx.com; the pin configuration on other AD9850 boards may be

different, so you must carefully examine your own AD9850 board before connecting it to the BeagleBone!

In order to calculate the tuning word for configuring the AD9850 output frequency, we need to know the relationship of the output frequency, reference clock, and tuning word of the AD9850. This is determined by the formula taken from the datasheet on AD9850 (**Fig.74)**.

$$f_{OUT} = (\Delta\ Phase \times CLKIN)/2^{32}$$

Fig.74

Here **ΔPhase** is the value of the 32-bit tuning word, **CLKIN** is the input reference clock frequency in MHz and $\mathbf{f_{OUT}}$ is the frequency of the output signal in MHz.

The following source code (**Listing 27**) allows to directly set the output frequency using the formula shown above.

Listing 27.

```
import Adafruit_BBIO.GPIO as GPIO
import math

FQ_UD = “P8_14”   # AD9850 FQ_UD pin is tied to P8_14
W_CLK = “P8_13”   # AD9850 W_CLK pin is tied to P8_13
DAT  =  “P8_15”   # AD9850 DATA pin is tied to P8_15

def setFreq(freq):
    freqWord = freq * pow(2, 32) / 125000000  # ref.clock = 125MHz
    mask = 0x1;
    w = 0;
    while (w < 32):  # writing bits W0—>W31
    GPIO.output(W_CLK, GPIO.LOW)
    GPIO.output(DAT, freqWord & mask)
```

```
    GPIO.output(W_CLK, GPIO.HIGH)
    freqWord = freqWord >> 1
    w += 1
    w = 32
    while (w < 40):  # writing the last byte = 0x0, bits W32—>W39
    GPIO.output(W_CLK, GPIO.LOW)
    GPIO.output(DAT, GPIO.LOW)
    GPIO.output(W_CLK, GPIO.HIGH)
    w += 1
    GPIO.output(FQ_UD, GPIO.HIGH)
```



```
GPIO.setup(FQ_UD, GPIO.OUT)  # set P8_14 as output
GPIO.setup(W_CLK, GPIO.OUT)  # set P8_13 as output
GPIO.setup(DAT, GPIO.OUT)       # set P8_15 as output

GPIO.output(W_CLK, GPIO.HIGH)
GPIO.output(W_CLK, GPIO.LOW)
GPIO.output(FQ_UD, GPIO.HIGH)
GPIO.output(FQ_UD, GPIO.LOW)
setFreq(27599);  # output frequency in Hz
```

In this source code, the **setFreq()** function sets the frequency of the sine wave output of AD9850. The function takes a single parameter which the desired frequency in Hz. When entered, the function calculates the binary code corresponding to the output frequency and stores the value in the **freqWord** variable by executing the following statement:

```
freqWord = freq * pow(2, 32) / 125000000
```

Then the **freqWord** bits are written to the AD9850 control registers using two **while** loops. The **W_CLK**, **FQ_UD** and **DAT** constants are assigned the pins being used in the SPI interface.

Processing DDS waveforms

The sine wave output of AD9850 can be used for creating a square wave signal by introducing the simple converter circuit shown in **Fig.75**.

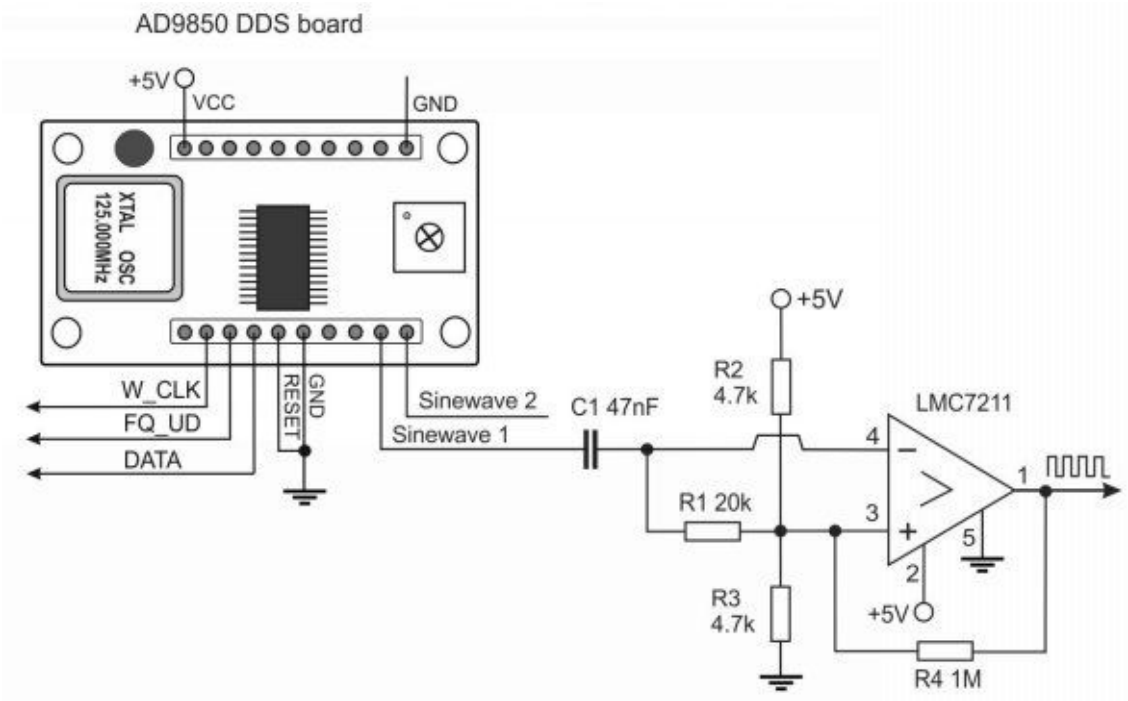


Fig.75

In this circuit, the sine wave signal (**Sinewave 1** output) is applied to the zero-crossing detector circuit with the comparator LMC7211 through the coupling capacitor C1. The TTL-compatible rectangular pulse train can be taken from pin 1 of the comparator.

The comparator circuit can also be powered by the lower voltage, say, +3.3V. Almost any general purpose comparator IC can be applied in this circuit, for instance, TLC3702, LM2903, etc. Note that for the higher frequencies it would be better to use some high-speed comparator, for example, MAX961.

One more circuit presented below allows to amplify the sine wave signal of AD9850. The output signal of the device has the amplitude around 0.6V, so a developer may want to amplify such signal up a few volts. The simple transistor amplifier (**Fig.76**) allows to obtain the amplitude of the output signal that is twice the amplitude of the input signal.

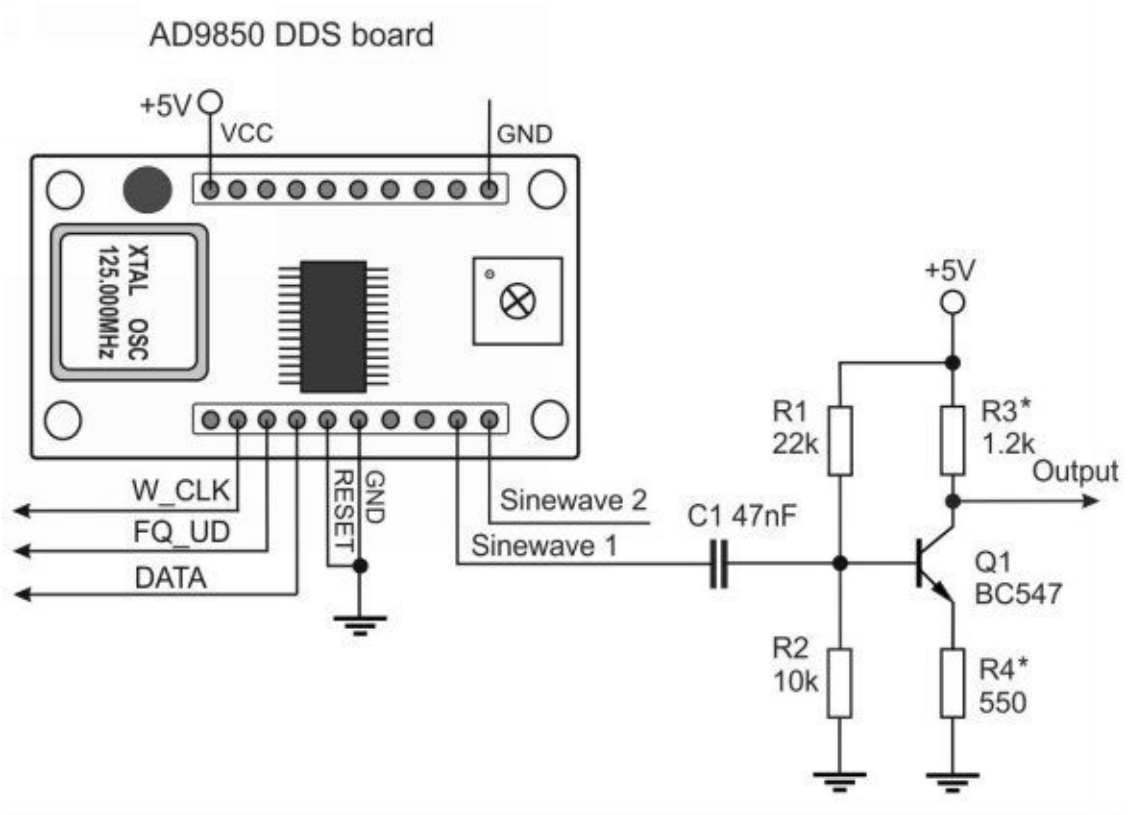


Fig.76

The amplifier circuit is built around the bipolar transistor BC547 (Q1) connected as a classic amplifier with a common emitter. The R1-R2 network provides the bias to Q1. The gain G of this circuit will approximately be calculated as:

G = R3/R4

In our case, G = 1.2/0.55 ≈ 2.2, so the input signal with the amplitude of 0.6V fed to the base of Q1 will be amplified to 0.6 x 2.2 ≈ 1.32V.

Using serial interfaces of the BeagleBone Black

Often a developer needs more capabilities than the BeagleBlack can provide. If, for example, you want to build a system which would measure parameters of input signals or process input/output signals in real time – you may need to employ some external microprocessor board capable of processing signals in real-time.

The good choices for performing such tasks are Arduino boards. We will discuss interfacing two popular boards, Arduino Uno and Arduino Due to the BeagleBone Black. The communications between Arduino boards and a BeagleBone Black can be implemented using a serial interface.

Before establishing the connection between Arduino and BeagleBone we should configure serial interfaces on the BeagleBone Black. The BeagleBone has six on-board serial ports, but only the /dev/ttyO0 is enabled by default and is coupled to the serial console. The other serial ports must be enabled before using them in projects.

The following sequence describes how to configure serial ports on the BeagleBone Black rev C.

First, we should edit the /boot/uboot/uEnv.txt file using the **vim** or **nano** editor. If you are logged in as root, use the command:

```
root@beaglebone:/# nano /boot/uboot/uEnv.txt
```

If not the root, use the following:

```
$ sudo nano /boot/uboot/uEnv.txt
```

The following line must be added to the /boot/uboot/uEnv.txt:

```
cape_enable=capemgr.enable_partno=BB-UART1,BB-UART2
```

Below is the content of the /boot/uboot/uEnv.txt after inserting the above line:

##Video: Uncomment to override:

##see: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation$

#kms_force_mode=video=HDMI-A-1:1024x768@60e

##Enable systemd

systemd=quiet init=/lib/systemd/systemd

##BeagleBone Cape Overrides

##BeagleBone Black:

##Disable HDMI/eMMC

#cape_disable=capemgr.disable_partno=BB-BONELT-HDMI,BB-BONELT-HDMIN,BB-BONE-EMMC-2G

##Disable HDMI

#cape_disable=capemgr.disable_partno=BB-BONELT-HDMI,BB-BONELT-HDMIN

##Audio Cape (needs HDMI Audio disabled)

#cape_disable=capemgr.disable_partno=BB-BONELT-HDMI

#cape_enable=capemgr.enable_partno=BB-BONE-AUDI-02

##Example

#cape_disable=capemgr.disable_partno=

cape_enable=capemgr.enable_partno=BB-UART1,BB-UART2

##WIP: v3.14+ capes

#cape=ttyO1

It is seen that the new line was added to the **Example** section. This enables the serial ports UART1 (/dev/ttyO1) and UART2 (/dev/ttyO2).

Then we have to reboot the system to make both added serial ports available to a developer. We can check this by entering:

```
root@beaglebone:/# ls -l /dev/ttyO*

crw-rw-- 1 root tty     248, 0 May 15 02:18 /dev/ttyO0
crw-rw-T 1 root dialout 248, 1 Jan  1  2000 /dev/ttyO1
crw-rw-T 1 root dialout 248, 2 Jan  1  2000 /dev/ttyO2
```

The table below shows the UART signal lines with associated BeagleBone Black pins:

	RX	**TX**	**CTS**	**RTS**	**Device**	**Notes**
UART0	J1_4	J1_5			/dev/ttyO0	
UART1	P9_26	P9_24	P9_20	P9_19	/dev/ttyO1	
UART2	P9_22	P9_21	P8_37	P8_38	/dev/ttyO2	
UART3		P9_42	P8_36	P8_34	/dev/ttyO3	TX only
UART4	P9_11	P9_13	P8_35	P8_33	/dev/ttyO4	
UART5	P8_38	P8_37	P8_31	P8_32	/dev/ttyO5	

To check UART1 and UART2 connect pins **P9.24** (UART1 TX) and **P9.22** (UART 2 RX), then open two terminal windows. In one of them enter:

```
root@beaglebone:/# cat < /dev/ttyO2
```

This command allows to receive the string on UART2. In the second window enter:

```
root@beaglebone:/# echo UART1-UART2 Test > /dev/ttyO1
```

This command sends the string out of UART1. After pressing Enter in other window you

should see:

root@beaglebone:/# cat < /dev/ttyO2

UART1-UART2 Test

Once the BeagleBone serial ports have been configured, we can use them in projects with Arduino boards.

Interfacing BeagleBone Black to Arduino Uno

Let’s begin with Arduino Uno. The Arduino IDE development environment for Arduino Uno can be installed on the BeagleBone Black, so a developer can build and debug the applications directly in Arduino IDE using the Processing language.

It is possible to install the Arduino IDE for Uno in various ways. One possible approach is described below.

First, update our system. When you are logged in as root enter:

```
root@beaglebone:/# apt-get update
```

or, as an ordinary user:

```
$ sudo apt-get update
```

Than we must install the additional libraries by executing:

```
$ sudo apt-get install avr-libc libftdi1 avrdude librxtx-java openjdk-6-jre
```

Note that some of libraries may already have been installed in your system. When this step is complete, we can install the Arduino IDE tool.

Assuming that the want to install the application into the directory ‘arduino’, execute the following sequence:

```
root@beaglebone:/# mkdir ~/arduino
root@beaglebone:/# cd ~/arduino
root@beaglebone:/# wget http://arduino.googlecode.com/files/arduino-1.0.5-linux64.tgz
root@beaglebone:/# tar xfz arduino-1.0.5-linux64.tgz
```

The Arduino IDE comes with a set of libraries meant for x86/x64 platform; those libraries are useless for the BeagleBone Black, so we need to replace them by arm-compatible versions. The arm libraries have already been installed, though they are located in various system directories. To use them in Arduino IDE we should make the following links by using the ln Linux command:

```
cd ~/arduino/arduino-1.0.5/lib/
ln -sf /usr/lib/jni/librxtxSerial.so librxtxSerial.so
ln -sf /usr/share/java/RXTXcomm.jar RXTXcomm.jar

cd ~/arduino/arduino-1.0.5/hardware/tools/
ln -sf /usr/bin/avrdude avrdude
ln -sf /etc/avrdude.conf avrdude.conf

cd ~/arduino/arduino-1.0.5/hardware/tools/avr/bin
ln -sf /usr/lib/avr/bin/ar avr-ar
ln -sf /usr/lib/avr/bin/as avr-as
ln -sf /usr/lib/avr/bin/ld avr-ld
ln -sf /usr/lib/avr/bin/nm avr-nm
ln -sf /usr/lib/avr/bin/objcopy avr-objcopy
ln -sf /usr/lib/avr/bin/objdump avr-objdump
ln -sf /usr/lib/avr/bin/ranlib avr-ranlib
ln -sf /usr/lib/avr/bin/strip avr-strip
ln -sf /usr/bin/avr-cpp avr-cpp
ln -sf /usr/bin/avr-g++ avr-g++
ln -sf /usr/bin/avr-gcc avr-gcc
```

When done, we can remove the bin.gcc directory which is now not needed:

```
rm -rf  /arduino/arduino-1.0.5/hardware/tools/avr/bin.gcc
```

At the next step we should install the **Ino** package itself using the following sequence:

```
mkdir ~/git/amperka
cd ~/git/amperka
git clone git://github.com/amperka/ino.git
cd ino
make
make install
sudo ln -s ~/arduino/arduino-1.0.5 /usr/share/arduino
```

To launch the Arduino IDE we should jump into the directory where the arduino executable is located and launch the program. On my BeagleBone that looks like the following:

```
root@beaglebone:/# ./arduino
```

While compiling the source code in Arduino IDE, we can get the following message:

Couldn’t determine program size: {0}

This, however, doesn’t affect the uploading a program.

Some words about hardware interfacing between Arduino Uno and BeagleBone Black.

Remember that Arduino-compatible boards with AVR microprocessor (Arduino Uno, Arduino Nano, Arduino Micro, etc.) all provide the 5V TTL levels for both digital inputs and outputs. When you directly connect BeagleBone GPIO pins to the Arduino ports, use bi-directional 3.3-5V converters so that to prevent your BeagleBone Black from damaging!

It is possible, however, to directly connect the BeagleBone Black GPIO pins configured as outputs, to the Arduino GPIO configured as inputs, because the 3.3V output signals from the BeagleBone Black will be properly processed by Arduino. Anyway, be very careful when assembling your project!

There a lot of information concerning Arduino Uno software development, so we will look one useful example where the BeagleBone Black and Arduino Uno operate together. Assume that we need to write 8-bit data at a time to external parallel interface. It is problematic to do that on the BeagleBone, but with Arduino Uno it would be extremely simple. The application running in the BeagleBone will send a command to the Arduino Uno via the serial interface. The hardware circuit of such system will look like the following (**Fig.77)**.

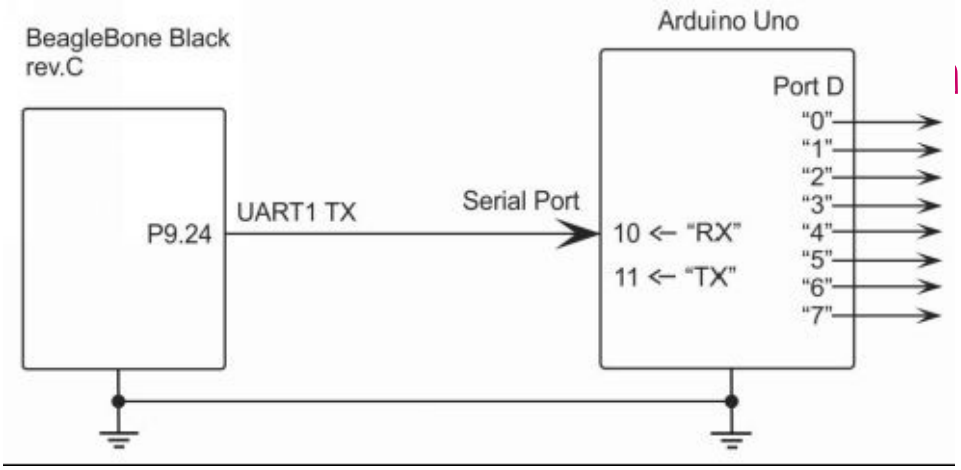


Fig.77

In this system, the command is transferred from the BeagleBone Black to Arduino, so only a **TX** signal line of the /dev/ttyO1 interface is used. On the Arduino side, the **RX** line assigned to pin "**10**" is used. **Note** that the same parameters for the serial communication should be set on both ends of the communication channel. The following projects use the common parameters for the serial interface: 9600 baud, 8 data bits, no parity check.

The Arduino Uno sketch may look like the following (**Listing 28**).

Listing 28.

```
#include <avr/io.h>
#include <SoftwareSerial.h>

// The application receives the command string via a Serial port,
// then parses it and set the value to the port D

SoftwareSerial Serial1(10,11); // RX-> 10, TX-> 11
String rcvSerial;
String cmd;    // the received command is saved here
String sVal;   // the port D data substring goes here

int dVal;      // an actual value
```

```
void parseStr(String s1)
{
int spaceIndex = s1.indexOf(‘ ‘);
cmd = s1.substring(0, spaceIndex);
cmd.trim();

if (cmd.equalsIgnoreCase(“portd”))
{
  int spaceIndex2 = s1.indexOf(‘ ‘, spaceIndex+1);
  sVal = s1.substring(spaceIndex, spaceIndex2);
  sVal.trim();
  dVal = sVal.toInt();
}
}

void setup()
{
  DDRD = 0xFF; // Port D (D0-D7) pins are all outputs
  PORTD = 0x0;
  Serial1.begin(9600);
}

void loop()
{
// put your main code here, to run repeatedly:
  if(Serial1.available() > 0)
  {
    rcvSerial = Serial1.readStringUntil(‘\r’);
    parseStr(rcvSerial);
    if (dVal >= 0 && dVal <= 255)
    PORTD = dVal;
```

```
  }
}
```


The Python application running on the BeagleBone Black has the following source code (**Listing 29**).

Listing 29.

```
import Adafruit_BBIO.UART as UART
import serial

UART.setup("UART1")
ser = serial.Serial(port = "/dev/ttyO1", baudrate = 9600)

if ser.isOpen():
    print "Setting the PORT D output"
    ser.write("PORTD 11")
ser.close()
```

Interfacing BeagleBone Black to Arduino Due

This section is dedicated to using a very powerful development board called Arduino Due in the BeagleBone projects. This board has a powerful Cortex-M3 processor and operates at high frequency (80 MHz) thus providing great capabilities for generating/measuring signals and implementing real-time algorithms. Programming this board is simple and requires Arduino IDE tool; while writing this book, the actual version 1.5.8 was accessible for free download on the Arduino site. The Arduino Due boards provide 3.3V TTL signals on their inputs/outputs, so we can directly connect GPIO pins of BeagleBone Black to the Arduino Due GPIO.

Unfortunately, the Arduino IDE for Due can't be installed on the BeagleBone Black easily, so we can develop the applications for Arduino Due on x86 platform (either Windows or Linux); nevertheless, we can use the serial interface to transfer data and commands between the BeagleBone Black and Arduino Due.

The serial communications between the BeagleBone Black and Arduino Due will look like that shown in **Fig.78**.

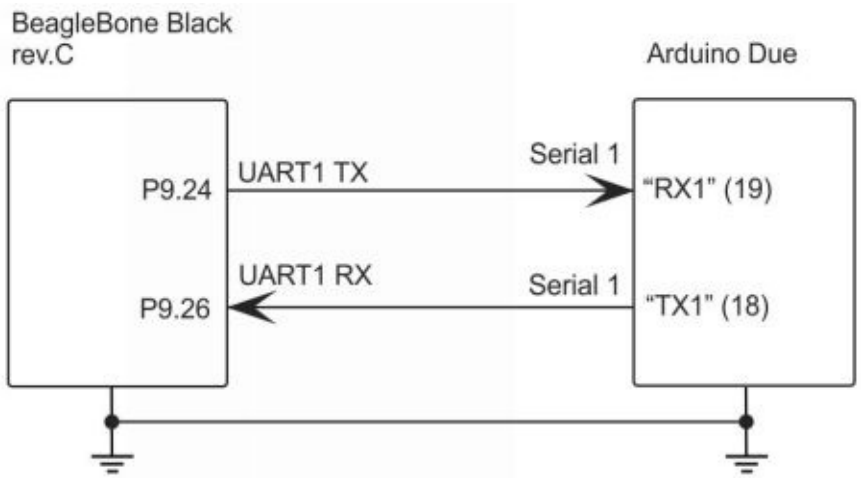


Fig.78

In this configuration, we can transfer commands to Arduino Due through pin **P9.24** of the BeagleBone Black and receive data from Arduino Due through pin **P9.26**. Both pins relate to UART1 (/dev/ttyO1).

On the Arduino Due side pin "**19**" ("**RX1**") allows to receive data through the **Serial1** interface, while the data stream is transferred through pin "**18**" ("**TX1**") associated with **Serial1**.

The following simple demo project illustrates transferring a text string from Arduino Due to the BeagleBone Black. In this project, data from pin "**TX1**" are transferred to pin **P9.26** of the BeagleBone.

The Arduino Due source code is shown in **Listing 30**.

Listing 30.

```
long int cnt = 0;
void setup() {
  // put your setup code here, to run once
Serial1.begin(9600); // TX1-RX1
}

void loop() {
    Serial1.print(++cnt);
  Serial1.println(": Arduino Due Serial 1 - BeagleBone UART1 Test");
  delay(3000);
}
```

This program simply sends the text string through the **Serial1** interface (pin "**TX1**") every 3 s. The BeagleBone Black source code written in Python looks like the following (**Listing 31**).

Listing 31.

```
import Adafruit_BBIO.UART as UART
import serial

UART.setup("UART1")
ser1 = serial.Serial(port = "/dev/ttyO1", baudrate = 9600)
```

```
cnt = 0
if ser1.isOpen():
    while cnt <= 10:
    bRead = ser1.inWaiting()
    if bRead > 0:
    str1 = ser1.readline()
    print str1
    cnt = cnt + 1
ser1.close()
```

The program periodically (variable **cnt**) reads the data being transferred via /dev/ttyO1 device (pin **P9.26**). When **cnt** exceeds 10, the application terminates.

While running, the Python program produces the following output:

```
root@beaglebone:/# python Read_UART1.py
1: Arduino Due Serial 1 - BeagleBone UART1 Test
2: Arduino Due Serial 1 - BeagleBone UART1 Test
3: Arduino Due Serial 1 - BeagleBone UART1 Test
4: Arduino Due Serial 1 - BeagleBone UART1 Test
5: Arduino Due Serial 1 - BeagleBone UART1 Test
6: Arduino Due Serial 1 - BeagleBone UART1 Test
7: Arduino Due Serial 1 - BeagleBone UART1 Test
8: Arduino Due Serial 1 - BeagleBone UART1 Test
9: Arduino Due Serial 1 - BeagleBone UART1 Test
10: Arduino Due Serial 1 - BeagleBone UART1 Test
11: Arduino Due Serial 1 - BeagleBone UART1 Test
```

Let’s discuss a few projects where BeagleBone Black and Arduino Due are used together.

Measuring the frequency of digital signals with Arduino Due

The BeagleBone Black allows to measure an amplitude of analog (continuous) signals by using a built-in analog-to-digital converter. When dealing with digital pulses, the amplitude is already predetermined by TTL-compatible levels. Meanwhile, other parameters of digital pulse trains (frequency, pulse width, duty cycle) should often be measured.

One of approaches for measuring these parameters is based upon using additional electronic circuits operating in real-time. The following two projects illustrate how to measure the frequency and pulse width of digital signals using the Arduino Due board.

In the first project, Arduino Due will measure the frequency of a digital pulse train and pass the result obtained to the BeagleBone Black through the serial interface.

The connections between the BeagleBone Black and Arduino Due are shown in **Fig.79.**

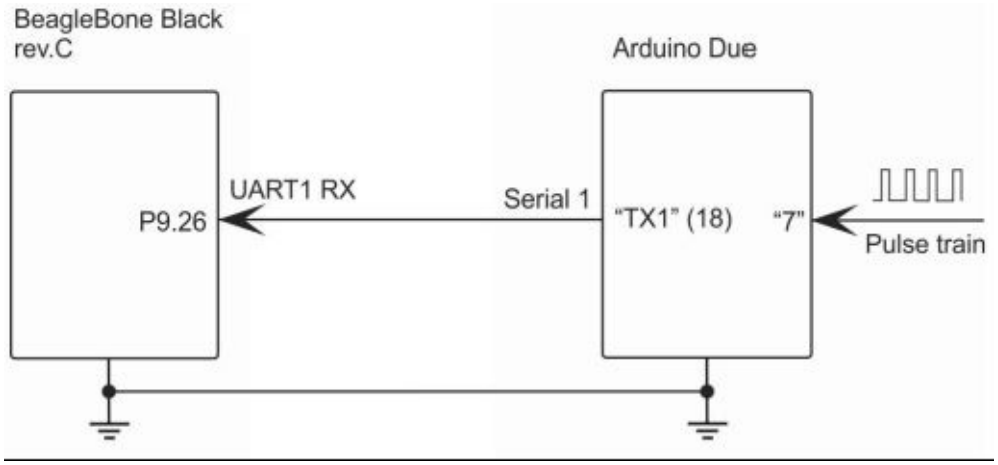


Fig.79

The Arduino Due source code is given in **Listing 32**.

Listing 32.

```
volatile long cnt = 0;
long tmp;
long num = 0;
```

```
void setup() {
  // set the digital pin as input
  pinMode(7, INPUT);
  attachInterrupt(7, fMeter, RISING); // Interrupt is triggered by the rising edge of a pulse
  Serial1.begin(9600);                    // arriving on pin 7
}


void loop()
{
cnt = 0;
delay(1000);
tmp = cnt;
Serial1.print(++num);
Serial1.print(": Input Frequency, Hz: ");
Serial1.println(tmp);
delay(3000);
};


void fMeter()
{
  cnt++;
}
```

The pulse train to be measured arrives on pin "**7"** of Arduino Due. To perform the measurement we configure this pin as an event source for the interrupt. The interrupt will be triggered by every rising edge of an incoming pulse. The interrupt service routine (ISR) **fMeter()** counts the rising edges during 1 s and saves the value in the **cnt** variable. When the time interval expires, the variable **tmp** will keep the number of pulses delivered to pin "**7"** per 1 s – this gives us the frequency of the input pulse train.

The measurements are repeated every 3 s. This application allows to reliably measure a frequency up to 90-110 KHz. The measured value is then transferred back to the BeagleBone Black via the serial interface (pin **P9.26**).

Measuring the pulse width of digital signals with Arduino Due

With the Arduino Due it is possible to measure the pulse width of digital signals. In this project we will use the same circuit as is shown in **Fig.79**. The signal to be measured arrives on pin "**7**" of Arduino Due.

The following Arduino Due source code (**Listing 33**) illustrates how to measure a pulse width.

Listing 33.

```
volatile int cnt = 0;
volatile int state;
volatile int tmp;

//TC1 of channel 0 is used

void TC3_Handler()
{
TC_GetStatus(TC1, 0);
cnt++;
}

void startTimer(Tc *tc, uint32_t channel, IRQn_Type irq, uint32_t frequency)
{
pmc_set_writeprotect(false);
pmc_enable_periph_clk((uint32_t)irq);
TC_Configure(tc, channel, TC_CMR_WAVE | TC_CMR_WAVSEL_UP_RC |
    TC_CMR_TCCLKS_TIMER_CLOCK1);
uint32_t rc = VARIANT_MCK/2/frequency;

TC_SetRA(tc, channel, rc/2); // the duty is 50%
```

```
TC_SetRC(tc, channel, rc);
TC_Start(tc, channel);
tc->TC_CHANNEL[channel].TC_IER=TC_IER_CPCS;
tc->TC_CHANNEL[channel].TC_IDR=~TC_IER_CPCS;
NVIC_EnableIRQ(irq);
}

void setup()
{
pinMode(7,INPUT);
startTimer(TC1, 0, TC3_IRQn, 1000000); //TC1 channel 0, the IRQ for that
    //channel and the frequency 1.0MHz ~ period of 1uS
Serial.begin(9600);
NVIC_DisableIRQ(TC3_IRQn);
}

void loop()
{
  tmp = 0;
  for (int i = 0; i< 51; i++)
  {
    cnt = 0;
    while ((state = digitalRead(7)) == 0x0);
    while ((state = digitalRead(7)) != 0x0);
    while ((state = digitalRead(7)) == 0x0);
    NVIC_EnableIRQ(TC3_IRQn);
    while ((state = digitalRead(7)) != 0x0);
    NVIC_DisableIRQ(TC3_IRQn);
    cnt = cnt - 4; //corection
    if (i > 0)
    tmp = tmp + cnt;
```

```
    delay(100);
  }
    int res = tmp/50 + 1;
    Serial.print(“Pulse Width, uS: “);
    Serial.println(res);
    delay(2000);
}
```

Control of PWM signals of Arduino Due using a serial interface

This project illustrates how to provide a remote control of the PWM signal generated by Arduino Due via a serial interface.

The circuit diagram of the project is shown in **Fig.80**.

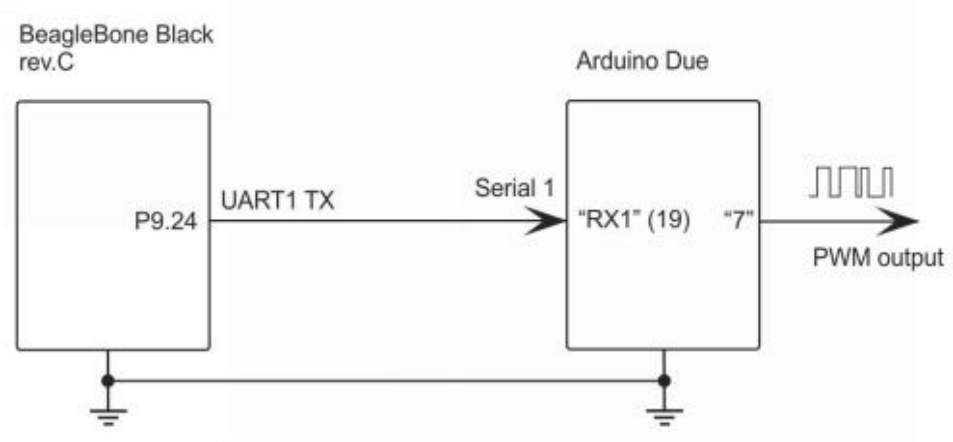


Fig.80

The BeagleBone Black passes the command via pin **P9.24** to the Arduino Due board. The application running on Due receives the command string via the **Serial1** port (pin "**19**"), then parses the command string and set the parameters (period and duty) of the PWM signal generated on pin "**7**" of Arduino Due.

The Arduino Due source code is shown below (**Listing 34**).

Listing 34.

```
#define PWM_CLOCK 4000000 // Clock frequency for PWM = 4.0MHz
#define CH  6             // PWM channel 6 (pin 7 of DUE) is selected

int PWM_PERIOD = 400; // the initial frequency is set to 10.0KHz (PWM_CLOCK / PWM_PERIOD)
int PWM_DUTY = 200;    // the initial duty = 50%
```

```
String rcvSerial;
String cmd;    // the received command is saved here
String sPeriod; // the period substring goes here
String sDuty;   // the duty substring goes here (0-255)

int period; // an actual period
int duty;   // an actual duty

void setup() {
  // put your setup code here, to run once
pinMode(7, OUTPUT); // PWML 6 is selected
analogWrite(7, 100);

PWMC_DisableChannel(PWM, CH);
PWMC_ConfigureClocks(PWM_CLOCK, 0, VARIANT_MCK); // Clock A is set to 4.0MHz, B = 0
PWMC_ConfigureChannel(PWM, CH,PWM_CMR_CPRE_CLKA,0,0);
PWMC_SetPeriod (PWM, CH, PWM_PERIOD);
PWMC_SetDutyCycle(PWM, CH, PWM_DUTY);
PWMC_EnableChannel(PWM,CH);

Serial1.begin(9600);
}

void parseStr(String s1)
{
int spaceIndex = s1.indexOf(‘ ‘);
cmd = s1.substring(0, spaceIndex);
cmd.trim();

if (cmd.equalsIgnoreCase(“pwm”))
{
```

```
  int spaceIndex2 = s1.indexOf(‘ ‘, spaceIndex+1);
  sPeriod = s1.substring(spaceIndex, spaceIndex2);
  sPeriod.trim();
  period = sPeriod.toInt();

  int spaceIndex3 = s1.indexOf(‘ ‘, spaceIndex+2);
  sDuty = s1.substring(spaceIndex3);
  sDuty.trim();
  duty = sDuty.toInt();
}
}

void loop() {
  // put your main code here, to run repeatedly:
  if(Serial1.available() > 0)
  {
    rcvSerial = Serial1.readStringUntil(‘\r’);
    parseStr(rcvSerial);
    if (duty > 0 && duty < 1000)
    {
    PWM_PERIOD = period;
    PWM_DUTY = duty;

    PWMC_DisableChannel(PWM, CH);
    PWMC_SetPeriod (PWM, CH, PWM_PERIOD);
    PWMC_SetDutyCycle(PWM, CH, PWM_DUTY);
    PWMC_EnableChannel(PWM,CH);
    }
  }
  delay(3000);
}
```

In this code, the section

```
PWMC_DisableChannel(PWM, CH);
PWMC_SetPeriod (PWM, CH, PWM_PERIOD);
PWMC_SetDutyCycle(PWM, CH, PWM_DUTY);
PWMC_EnableChannel(PWM,CH);
```

determines the period and duty cycle of the PWM signal of the channel 6. The signal appears on pin “**7**” of Arduino Due.

The statement

```
rcvSerial = Serial1.readStringUntil(‘\r’);
```

moves the string obtained from the serial interface to the **rcvSerial** variable. The next is the

```
parseStr(rcvSerial);
```

statement which calls the **parseStr()** function to parse the received string. The values of a period and duty cycle are saved in the **period** and **duty** variables, respectively.

The Python application running on the BeagleBone Black is presented by the following source code (**Listing 35**).

Listing 35.

```
import Adafruit_BBIO.UART as UART
import serial

UART.setup(“UART1”)
ser = serial.Serial(port = “/dev/ttyO1”, baudrate = 9600)
```

```
if ser.isOpen():
    print “Setting parametrs for the PWM signal”
    ser.write(“PWM 240 30”)
ser.close()
```

In the given case, the code sets the frequency of the PWM signal to be 4000000 / 240 = 16666 Hz. The duty cycle will be equal to 30 / 240 = 0.125 (12.5%).

Control of sawtooth signals of Arduino Due using a serial interface

This project illustrates how to implement the remote control of a sawtooth clock source generated by Arduino Due. Here the frequency of a sawtooth signal is determined by the BeagleBone Black application. The command string to Arduino Due is transferred from the BeagleBone through a serial interface. The sawtooth signal appears on the **DAC0** output of Arduino Due.

The schematic circuit of this project is shown in **Fig.81**.

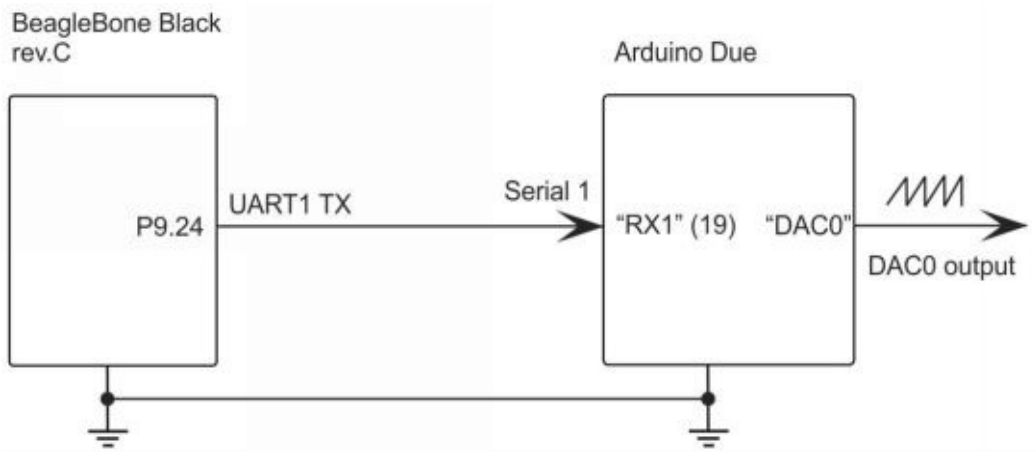


Fig.81

The Arduino Due source code is shown in **Listing 36**.

Listing 36.

```
const int NUM_SAMPLES = 200;
volatile int cnt = 0;

String rcvSerial;
String cmd;      // the received command is placed here
String sFreq;    // the frequency is placed here

int freq;        // an actual frequency in Hz
```

```
// The Timer TC0 Channel 1 is taken

void TC1_Handler()  // IRQ handler for Channel 1 of TC0
{
TC_GetStatus(TC0, 1);
analogWrite(DAC0, cnt++);   // use DAC0
if (cnt == NUM_SAMPLES-1)cnt = 0;
}

void startTimer(Tc *tc, uint32_t channel, IRQn_Type irq, uint32_t frequency)
{
pmc_set_writeprotect(false);
pmc_enable_periph_clk((uint32_t)irq);
TC_Configure(tc, channel, TC_CMR_WAVE | TC_CMR_WAVSEL_UP_RC |
    TC_CMR_TCCLKS_TIMER_CLOCK1);

uint32_t rc = VARIANT_MCK/2/frequency; // 2 was taken by default if we selected
    //TIMER_CLOCK1 above

TC_SetRC(tc, channel, rc);
TC_Start(tc, channel);
tc->TC_CHANNEL[channel].TC_IER=TC_IER_CPCS;
tc->TC_CHANNEL[channel].TC_IDR=~TC_IER_CPCS;
NVIC_EnableIRQ(irq);
}

void setup() {
  // put your setup code here, to run once
analogWriteResolution(8); // 8-bit resolution
startTimer(TC0, 1, TC1_IRQn, 200000); // the TC0 channel 1, the IRQ for this
```

```
    // channel and the desired frequency
    // the frequency at the DAC 0 output will be equal to timer frequency divided
    // by the selected number of samples
    // In this example, the frequency will be
    // 200000 / 200 = 1000 Hz

Serial1.begin(9600);
}

void parseStr(String s1)
{
int spaceIndex = s1.indexOf(‘ ‘);
cmd = s1.substring(0, spaceIndex);
cmd.trim();

if (cmd.equalsIgnoreCase(“sawtooth”))
{
  int spaceIndex2 = s1.indexOf(‘ ‘, spaceIndex+1);
  sFreq = s1.substring(spaceIndex, spaceIndex2);
  sFreq.trim();
  freq = sFreq.toInt();
}
}

void loop() {
  // put your main code here, to run repeatedly:
  if(Serial1.available() > 0)
  {
    rcvSerial = Serial1.readStringUntil(‘\r’);
    parseStr(rcvSerial);
    if (freq > 0 && freq < 1000)
```

```
    {
    NVIC_DisableIRQ(TC1_IRQn);
    freq = freq * 200; // the timer interrupt frequency should be set 200 times greater
    startTimer(TC0, 1, TC1_IRQn, freq);
    }
  }
  delay(3000);
}
```

The Python application running on the BeagleBone Black is presented by the following source code (**Listing 37**).

Listing 37.

```
import Adafruit_BBIO.UART as UART
import serial

UART.setup("UART1")
ser = serial.Serial(port = "/dev/ttyO1", baudrate = 9600)

if ser.isOpen():
    print "Setting parametrs for the SAWTOOTH signal"
    ser.write("SAWTOOTH 677")
ser.close()
```

With this code the frequency of the sawtooth signal on the **DAC0** output of Arduino Due is set to 677 Hz.

Wireless systems using Wixel Programmable USB Wireless Modules: project 1

The BeagleBone Black can be easily fit into a wireless measurement system using a Wixel Programmable USB Wireless Module. This section illustrates the design of the wireless system where the BeagleBone Black and Arduino Due work together using a wireless communication channel provided by the Wixel Programmable USB Wireless module.

Below there is a brief description of the Wixel Programmable USB Wireless module. The detailed information about this module can be found on www.pololu.com.

The Pololu Wixel is a general-purpose programmable module featuring a 2.4 GHz radio and USB. The Wixel is based on the CC2511F32 microcontroller from Texas Instruments, which has an integrated radio transceiver, 32 KB of flash memory, 4 KB of RAM, and a full-speed USB interface. A total of 15 general-purpose I/O lines are available, including 6 analog inputs.

There may three main ways to use this general-purpose module:

1. Directly connect the Wixel to a PC to create a wireless USB dongle.
2. Add USB connectivity to a project via the Wixel.
3. Add wireless capabilities to a remote, self-powered device.

The block diagram of this module is shown in **Fig.82**.

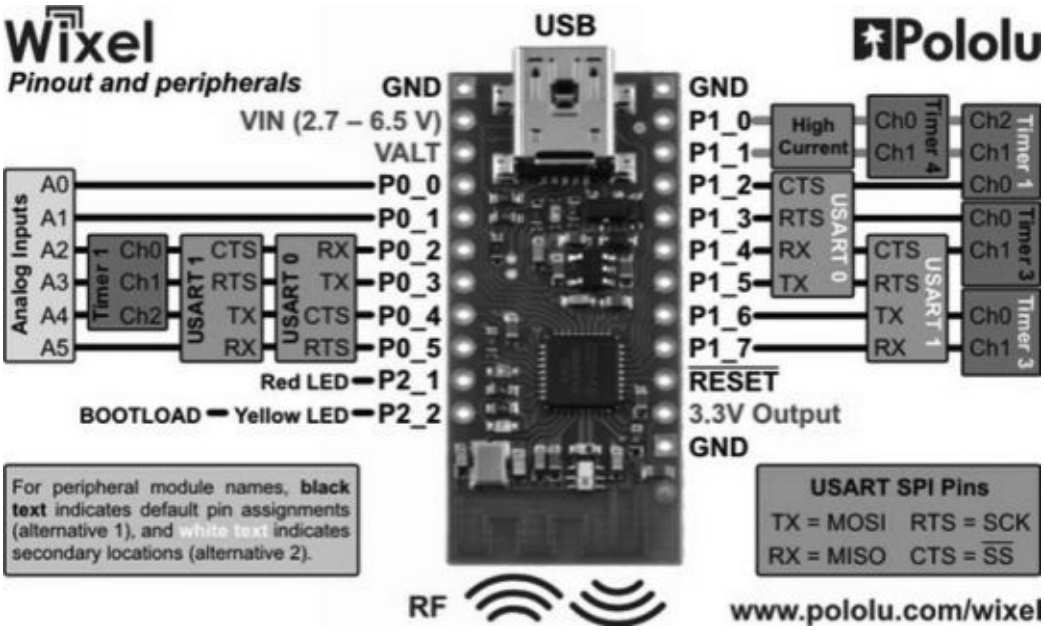


Fig.82

The Wixel software includes a built-in USB bootloader that can be used in conjunction with the free Wixel Configuration Utility to upload custom programs or precompiled,

open-source applications to the Wixel (no an external programmer is required).

No programming experience or compiler software is required to use these applications. A developer can simply download a different application to reuse the Wixel Wireless module in the new project.

Warning about radio regulations: The Wixel has not been tested or certified for conformance with any radio regulations, and the Wixel is shipped with only a bootloader that does not use the radio. The 2.4 GHz band is relatively unrestricted in many parts of the world, but it is your responsibility to comply with your local regulations if you program your Wixel to use its wireless capabilities.

The block diagram of a demo wireless system comprising the BeagleBone Black, Arduino Due and Wixel Wireless modules is shown in **Fig.83**.

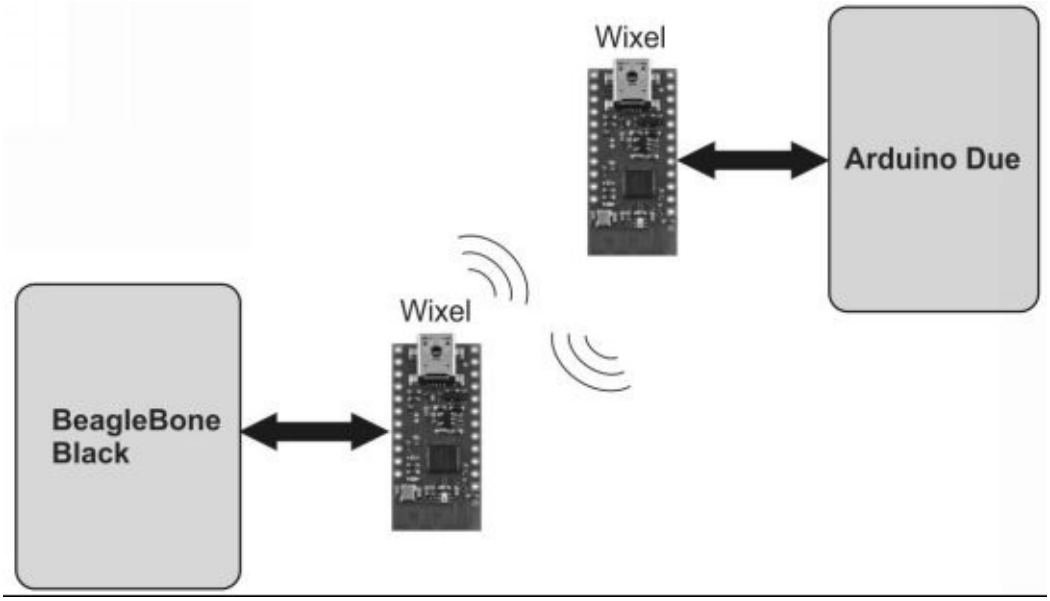


Fig.83

In this system, both Wixel modules provide wireless communication channel between the BeagleBone and Arduino Due. To use Wixel Wireless modules in such configuration, a **Wireless Serial Application** provided by Pololu should be installed on both modules.

Below is a short excerpt from the **Pololu Wixel User's Guide** concerning the use of the **Wireless Serial App**.

The **Wireless Serial App** allows you to connect two Wixels together to make a wireless, bidirectional, lossless serial link. It uses an RF bit rate of 350 kbps, is capable of carrying up to 10 KB of payload data per second, and can reach a range of approximately 50 feet (under typical conditions indoors). You can also use it to turn one Wixel into a USB-to-

TTL serial adapter. This app can run on multiple pairs of Wixels as long as each pair operates on a different radio channel (the channels should be separated by 2 to avoid interference).

This app is designed for pairs of Wixels; it will not work properly if three or more Wixels are broadcasting on the same radio channel.

To install the **Wireless Serial App** we should download the **Wireless Serial App (v1.3)** from http://www.pololu.com/file/download/wireless-serial-v1.3.wxl?file_id=0J484, open it with the **Wixel Configuration Utility**, choose the parameters or leave the default settings, and write the applications to two Wixel Wireless modules.

The wxl-file of the application can also be found in the EXAMPLES directory of the **Wixel Configuration Utility**. In Windows, the path will be as follows:

C:\Program Files\Pololu\Wixel\EXAMPLES\wireless-serial-v1.3.wxl

When the **Wireless Serial App** has been installed on the Wixel Wireless module, we can select any of three basic serial modes. In **USB-to-Radio** mode (1), the serial bytes from the USB virtual COM port get sent to the radio and vice versa. In **UART-to-Radio** mode (2), the serial bytes from the UART's RX line get sent to the radio and bytes from the radio get sent to the UART's TX line.

In **USB-to-UART** mode (3), the Wixel module acts like a normal USB-to-TTL serial adapter; bytes from the virtual COM port get sent on the UART's TX line and bytes from the UART's RX line get sent to the virtual COM port.

We can select which serial mode to use by setting the **serial_mode** parameter to the appropriate number (1 through 3) or we can leave the serial mode at 0 (which is the default). If the **serial_mode** is 0, then the Wixel Wireless module will automatically choose a serial mode based on how it is being powered as described in the table below, and it will switch between the different serial modes on the fly.

Power Source	**Serial Mode**	**Auto-Detect Serial Mode (serial_mode = 0)**
USB only	USB-to-Radio	
VIN only	UART-to-Radio	
USB and VIN	USB-to-UART	

Caution: The Wixel's I/O lines are not 5V tolerant. You must use level-shifters, diodes, or voltage dividers to connect the Wixel to outputs from 5V systems. Also, avoid drawing more current from an I/O line than it can provide. Avoid connecting multiple output pins together.

The serial data format used by this app is 8 data bits, one stop bit, with no parity, which is often expressed **8-N-1**. The data is non-inverted, so 0 V represents 0 and 3.3 V represents 1.

The following two projects use only two signal lines, TX and RX. The TX line is connected to pin **P1_6** pin of Wixel for transmitting serial data (0–3.3 V). The RX line is connected to pin **P1_7** of Wixel when receiving serial data (0–3.3 V, not 5 V tolerant). The Wixel Wireless modules used in our projects operate in mode 0 (**serial_mode** = 0). Both Wixel Wireless modules are powered from a stand-alone (external) +3.3V DC voltage source through the **VIN** pin, therefore they operate as **UART-to-Radio** bridges.

Our first demo project illustrates the design of the wireless system capable of adjusting the PWM duty cycle on the remote Arduino Due controller. This system will consist of two parts; one will use the BeagleBone Black to wirelessly transfer the parameters of a PWM signal, while the other part using Arduino Due will receive the data from the BeagleBone and set the parameters of a PWM signal accordingly.

The BeagleBone part of the wireless system is configured as is shown in the following block diagram (**Fig.84**).

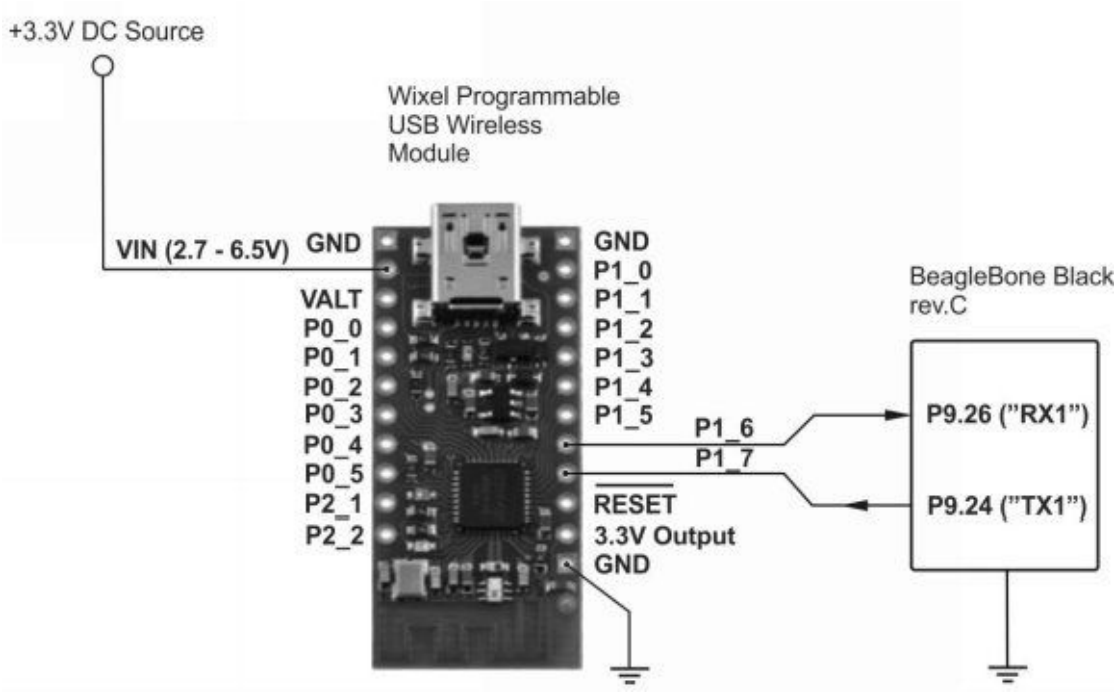


Fig.84

In this circuit, the power to the Wixel Wireless module is fed from the stand-alone +3.3V DC voltage source. The **P1_6** pin (TX) of the Wixel module is connected to pin **P9.26** (RX1) of the BeagleBone Black. The **P1_7** pin (RX) of the Wixel module is connected to pin **P9.24** (TX1) of the BeagleBone. The “ground” pins (GND) of both modules should be tied together.

The Arduino Due part of the system looks like the following (**Fig.85**).

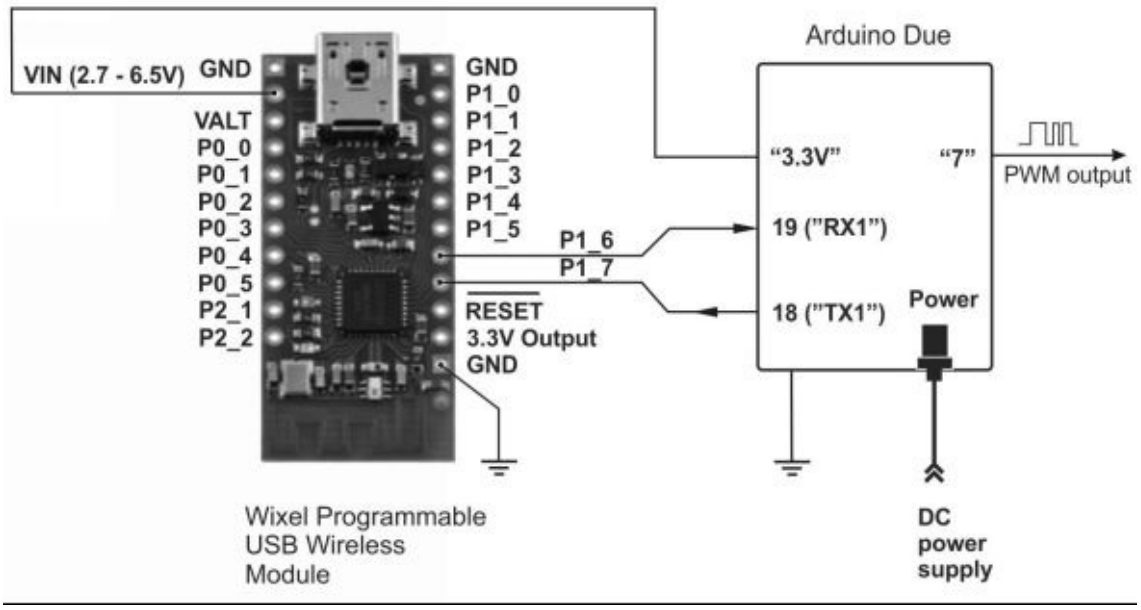


Fig.85

In this circuit, the power to the Wixel Wireless module is fed from the Arduino Due pin “**3.3V**”. The **P1_6** pin (TX) of the Wixel module is connected to pin “**19**” (“**RX1**”) of the Arduino Due board. The **P1_7** pin (RX) of the Wixel module is connected to pin “**18**” (“**TX1**”) of the Arduino Due. The “ground” pins (GND) of both modules should be tied together.

The BeagleBone Python source code is shown in **Listing 38**.

Listing 38.

```
import Adafruit_BBIO.UART as UART
import serial
```

```
UART.setup("UART1")
ser = serial.Serial(port = "/dev/ttyO1", baudrate = 9600)

if ser.isOpen():
    print "Setting parameters for the PWM signal"
    ser.write("PWM 100 72") # the base frequency of PWM = 100.0KHz/100 = 1KHz, duty = 72%
ser.close()
```

The Arduino Due source code is presented in **Listing 39**.

Listing 39.

```
// The application receives the command string via Serial1 port,
// then parses it and sets the parameters (period and duty)
// of the PWM signal (pin 7) of Arduino Due

#define PWM_CLOCK 100000 // Clock frequency for PWM = 100.0KHz
#define CH  6            // Determines the PWM channel 7(6) (pin 6(7) of DUE)

int PWM_PERIOD = 100; // the initial frequency is set to 1.0KHz (PWM_CLOCK / PWM_PERIOD)
int PWM_DUTY = 50;    // the initial duty = 50%

String rcvSerial;
String cmd;    // the received command is saved here
String sPeriod; // the period substring goes here
String sDuty;   // the duty substring goes here

int period; // an actual period
int duty;   // an actual duty
```

```
void setup() {
  // put your setup code here, to run once:
```

```
pinMode(7, OUTPUT); // PWML 6 is selected
analogWrite(7, 50);

PWMC_DisableChannel(PWM, CH);
PWMC_ConfigureClocks(PWM_CLOCK, 0, VARIANT_MCK); // Clock A is set to 100.0KHz, B = 0
PWMC_ConfigureChannel(PWM, CH,PWM_CMR_CPRE_CLKA,0,0);
PWMC_SetPeriod (PWM, CH, PWM_PERIOD);
PWMC_SetDutyCycle(PWM, CH, PWM_DUTY);
PWMC_EnableChannel(PWM,CH);

Serial1.begin(9600);
}

void parseStr(String s1)
{
int spaceIndex = s1.indexOf(' ');
cmd = s1.substring(0, spaceIndex);
cmd.trim();

if (cmd.equalsIgnoreCase("pwm"))
{
  int spaceIndex2 = s1.indexOf(' ', spaceIndex+1);
  sPeriod = s1.substring(spaceIndex, spaceIndex2);
  sPeriod.trim();
  period = sPeriod.toInt();

  int spaceIndex3 = s1.indexOf(' ', spaceIndex+2);
  sDuty = s1.substring(spaceIndex3);
  sDuty.trim();
```

```
  duty = sDuty.toInt();
}
}

void loop() {
  // put your main code here, to run repeatedly:
  if(Serial1.available() > 0)
  {
    rcvSerial = Serial1.readStringUntil('\r');
    parseStr(rcvSerial);
    if (duty > 0 && duty < 100)
    {
    PWM_PERIOD = period;
    PWM_DUTY = duty;

    PWMC_DisableChannel(PWM, CH);
    PWMC_SetPeriod (PWM, CH, PWM_PERIOD);
    PWMC_SetDutyCycle(PWM, CH, PWM_DUTY);
    PWMC_EnableChannel(PWM,CH);
    }
  }
  delay(3000);
}
```

Wireless systems using Wixel Programmable USB Wireless Modules: project 2

This project illustrates the design of the wireless measurement system where the low-cost Tiva™ C Series TM4C123G LaunchPad Evaluation Board from Texas Instruments is employed as an “intellectual” node. The Tiva C LaunchPad will process an analog signal from a sensor and transfer the result to the BeagleBone Black through the Wixel USB Wireless module.

A brief description of the Tiva C Series LaunchPad taken from the User’s Guide on this board is presented below.

The Tiva™ C Series TM4C123G LaunchPad Evaluation Board (EK-TM4C123GXL) is a low-cost evaluation platform for ARM Cortex™-M4F-based microcontrollers. The Tiva C Series LaunchPad design highlights the TM4C123GH6PMI microcontroller USB 2.0 device interface, hibernation module, and motion control pulse-width modulator (MC PWM) module. The Tiva C Series LaunchPad also features programmable user buttons and an RGB LED for custom applications. More details concerning Tiva™ C Series TM4C123G LaunchPad can be found in the user’s guide on this board. **Fig.86** shows a photo of the Tiva C Series LaunchPad.

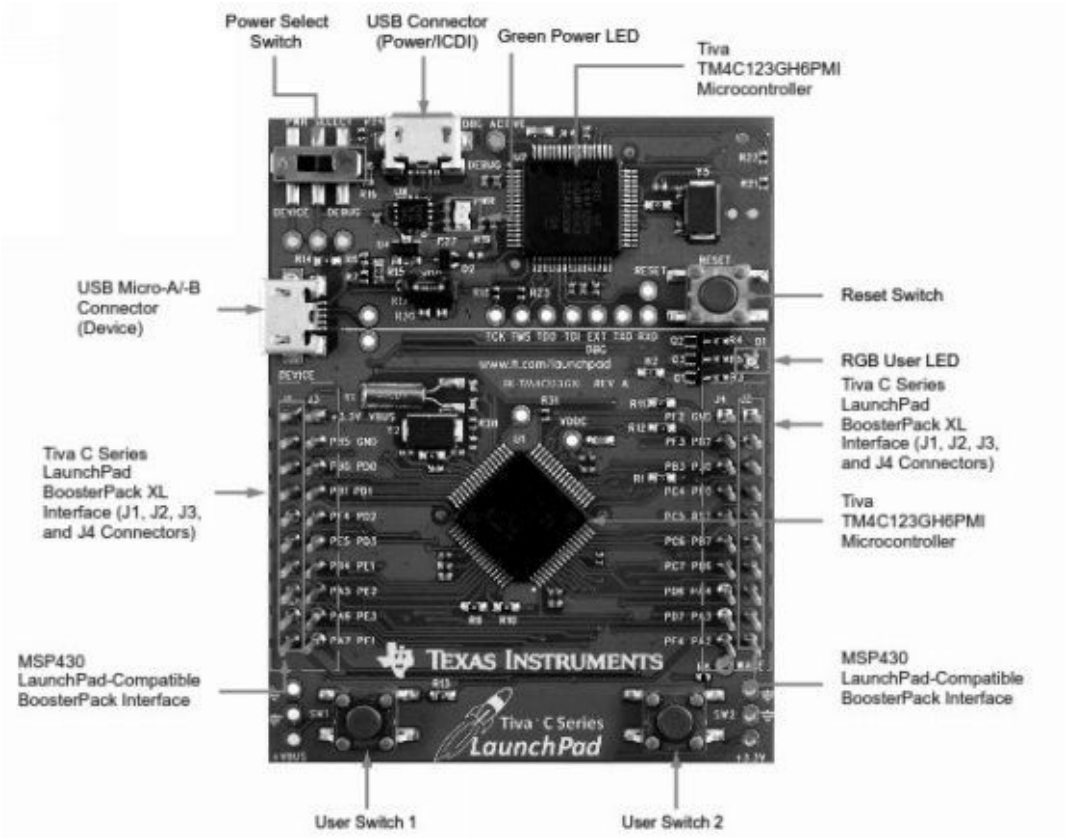


Fig.86

Our system will comprise the BeagleBone part whose block diagram is shown in **Fig.87** and the Tiva C LaunchPad part configured as is shown in **Fig.88**.

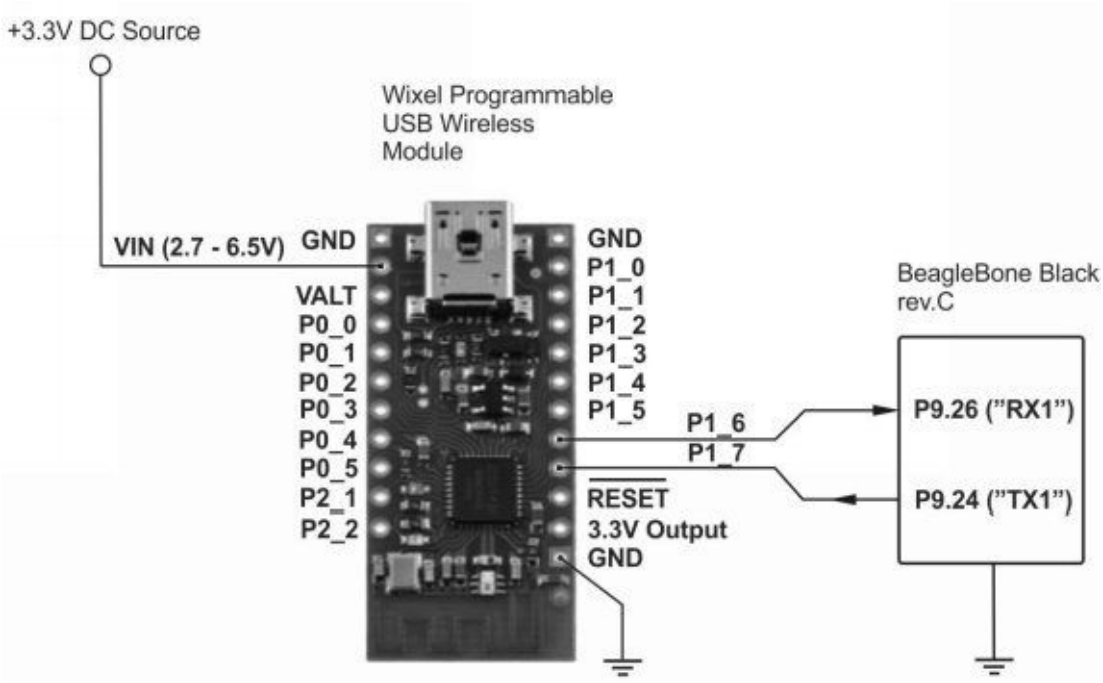


Fig.87

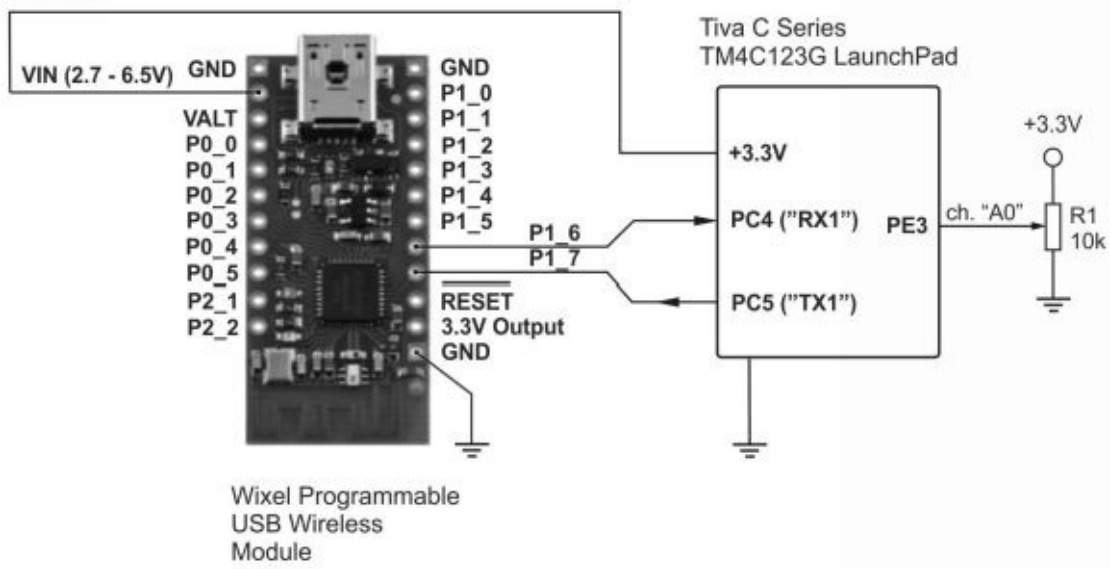


Fig.88

In the Tiva C LaunchPad part, the power to the Wixel Wireless module is fed from the pin "**+3.3V**" on the Tiva C LaunchPad board. The **P1_6** pin (TX) of the Wixel module is connected to pin **PC4** (RX1) of the Tiva C LaunchPad. The **P1_7** pin (RX) of the Wixel module is connected to pin **PC5** (TX1) of the Tiva C. The "ground" pins (GND) of both modules should be tied together.

The analog signal is simulated by the voltage divider formed by the potentiometer R1. The

signal from the wiper of the potentiometer arrives on the channel **A0** (pin **PE3**) of the Tiva C LaunchPad.

The result of the measurement of the analog input voltage is then transferred to the BeagleBone Black via the Wixel Wireless module.

The Python application running on the BeagleBone Black has the following source code (**Listing 40**).

Listing 40.

```
import Adafruit_BBIO.UART as UART
import serial

UART.setup("UART1")
ser1 = serial.Serial(port = "/dev/ttyO1", baudrate = 9600)

cnt = 0
if ser1.isOpen():
    while cnt <= 10:
    bRead = ser1.inWaiting()
    if bRead > 0:
    str1 = ser1.readline()
    print str1
    cnt = cnt + 1
ser1.close()
```

The application performs 10 iterations while receiving data via the /dev/ttyO1 serial device.

The Tiva C source code developed in the **Energia IDE** is presented in **Listing 41**.

Listing 41.

```
int iPin = A0;    // select the input pin for the analog signal
int binVal = 0;
float VREF = 3.34; // the measured value corresponding to VDD
float res;

void setup()
{
  // Open serial communications and wait for port to open:
  Serial1.begin(9600); // RX1 —> PC_4, TX1—>PC_5
     }

void loop() // run over and over
{
  binVal = analogRead(iPin);
  res = VREF /4096.0 * binVal;
  Serial1.print("A0 input, V: ");
  Serial1.println(res, 3);
  delay(3000);
}
```

The built-in A/D converter of Tiva C LaunchPad provides the 12-bit resolution; the full-scale range of the signal being measured is determined by the inner reference voltage of 3.3V (this is reflected by the **VREF** variable).

BeagleBone Black in wireless systems using Bluetooth

Simple wireless measurement and control systems based upon the BeagleBone Black can be built using a Bluetooth protocol. Such systems besides the BeagleBone may comprise desktop PCs and microcontroller boards. In this section we will discuss simple wireless systems consisting of two “intellectual” nodes, one of them will be the BeagleBone Black itself and other will be a desktop PC running Linux.

In this configuration, the desktop PC running Linux can act as a host machine which performs high-level processing of data obtained from the remote BeagleBone Black module. The BeagleBone Black will pick up signals from various sensors, convert them into a suitable format and transfer data to the host PC. Both a desktop PC and BeagleBone Black will use USB Bluetooth devices to communicate wirelessly.

In software, such systems can be arranged in a “client-server” configuration. In this case, the PC running Linux acts as a “Bluetooth server”, while the BeagleBone Black will run the “Bluetooth client” application. The block diagram of such a system may look like the following (**Fig.89)**.

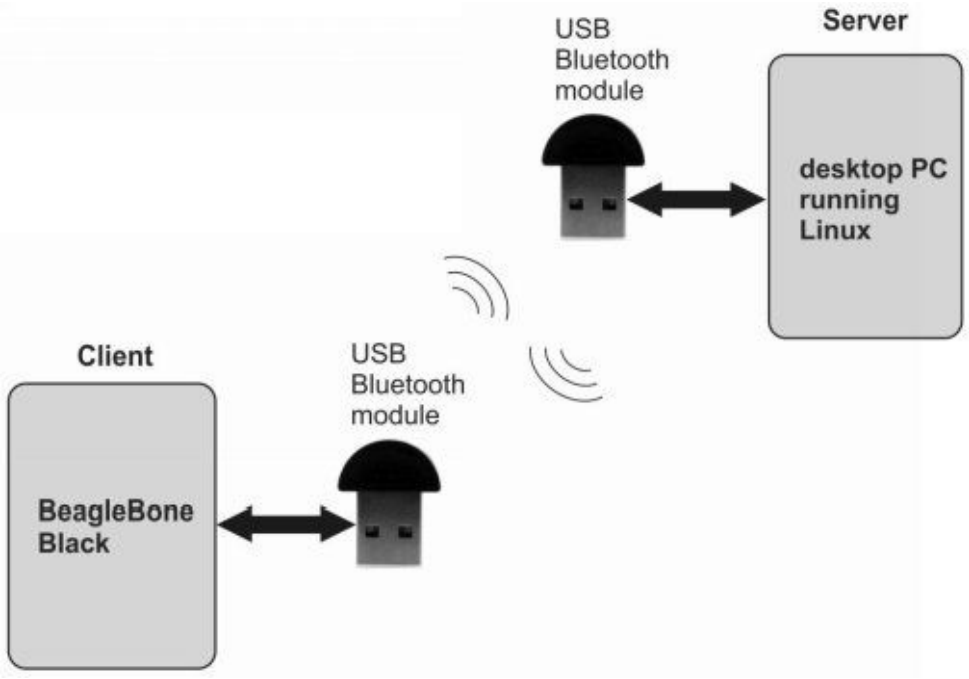


Fig.89

The configuration shown in **Fig.89** will be used for building a demo project a bit later in this section. Before we start off our design, we must configure software on both Linux machines in order to develop Bluetooth applications. There are numerous on-line documents relating to this theme, so we will consider only a few basic steps.

Since we will develop both client and server applications in Python, we need to install the **python-bluez** software module first. If you logged in as a superuser root, enter:

```
root@beaglebone:/# apt-get update
root@beaglebone:/# apt-get install python-bluez
```

If you are logged in as an ordinary user, enter:

```
$ sudo apt-get update
$ sudo apt-get install python-bluez
```

Once we have the Bluetooth software installed, let's detect USB Bluetooth devices connected to our system and determine their parameters. The Bluetooth device on my desktop PC running Linux Mint was detected as:

```
root@beaglebone:/# hcitool dev
Devices:
        hci0            00:15:83:15:A3:10
```

The **hcitool** command returns the list of Bluetooth devices available in system. If we need much more information about Bluetooth device, we can obtain it using **–a** option. For example, the detailed information on the hci0 device appears as follows:

```
root@beaglebone:/# hciconfig -a hci0
hci0:   Type: BR/EDR  Bus: USB
    BD Address: 00:15:83:3D:0A:57  ACL MTU: 310:10  SCO MTU: 64:8
    UP RUNNING PSCAN ISCAN
    RX bytes:1249 acl:0 sco:0 events:45 errors:0
    TX bytes:452 acl:0 sco:0 commands:44 errors:0
    Features: 0xff 0xff 0x8f 0xfe 0x9b 0xf9 0x00 0x80
    Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
    Link policy: RSWITCH HOLD SNIFF PARK
    Link mode: SLAVE ACCEPT
    Name: 'beaglebone-0'
```

Class: 0x520100

Service Classes: Networking, Object Transfer, Telephony

Device Class: Computer, Uncategorized

HCI Version: 2.0 (0x3) Revision: 0xc5c

LMP Version: 2.0 (0x3) Subversion: 0xc5c

Manufacturer: Cambridge Silicon Radio (10)

Note that Bluetooth devices installed in your system may have other parameters, of course.

Assuming that a Bluetooth adapter is present on a remote BeagleBone Black board, try to reach this device from our desktop PC by entering:

```
root@beaglebone:/# hcitool scan
Scanning …
        9C:02:98:1E:73:6B       S5610
        00:15:83:3D:0A:57       beaglebone-0
```

It is seen that two remote Bluetooth devices were detected. One of them, named “beaglebone-0”, was connected to the BeagleBone Black; this device will be used for data transfer from the BeagleBone Black to the PC running Linux Mint OS.

Alternatively, we can also check the presence of the device “beaglebone-0” by entering the command “**hcitool dev**” in the terminal window on the BeagleBone Black.

Once Bluetooth devices operate properly on both desktop PC and BeagleBone Black, we can test the data flow through a wireless communication channel.

To do that, let’s develop the simple Bluetooth server application which will be running on the desktop PC and the Bluetooth client launched on the BeagleBone Black. The Bluetooth server when launched enters the endless loop waiting for data to be sent by the Bluetooth client.

The client application will periodically send a text string to the Bluetooth server.

The Python source code for a Bluetooth server is shown in **Listing 42**.

Listing 42.

```
#A simple Bluetooth server to receive messages from a Bluetooth client

import signal
import sys
import bluetooth

def signal_handler(signal, frame):
    print 'Terminating on Ctrl+C…'
    client.close()
    s.close()
    sys.exit(0)

hostMACAddress = '00:15:83:15:A3:10' # The MAC address of a Bluetooth
                                     # adapter on the server.

port = 6                    # 6 is an arbitrary choice.
                            # However, it must match the port
                            # used by the client.

backlog = 1
size = 1024
s = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
s.bind((hostMACAddress,port))
s.listen(backlog)
signal.signal(signal.SIGINT, signal_handler)

print "Waiting for incoming requests from Bluetooth clients on port 6…"
print "Press Ctrl+C for exit…"

while 1:
    client, address = s.accept()
    data = client.recv(size)
    if data:
        print(data)
    client.close()
```

Note that the Bluetooth server application operates much like the TCP/IP server. The Bluetooth server uses the listening Bluetooth socket to wait for incoming requests from Bluetooth clients. The listening socket (variable **s**) is created and configured by the following sequence:

```
s = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
s.bind((hostMACAddress,port))
s.listen(backlog)
```

Here the hostMACAddress is assigned the Bluetooth address of the server. The variable port holds the number of the port (this can be a random value, 6 in our case).

Once the listening socket s has been created, the program enters the endless while 1 loop which continues until the Ctrl+C is pressed. If a new request comes from a Bluetooth client, the program code creates a working connection (“working socket”) by the statement:


```
client, address = s.accept()
```

Here the variable **client** identifies the connection established with the particular Bluetooth client. Then the program receives the data and, if not empty, outputs it in the terminal window:

```
data = client.recv(size)
      if data:
          print(data)
```

Finally, the working connection should be closed by invoking:

```
client.close()
```

To terminate the Bluetooth server, we use the Ctrl+C combination. In order to terminate the running process correctly, we must configure the signal handler for Ctrl+C. First, define a signal handler **signal_handler** using the following statements:

```
def signal_handler(signal, frame):
     print ‘Terminating on Ctrl+C…’
     client.close()
     s.close()
     sys.exit(0)
```

The signal_handler prints a message, closes the listening and working sockets and then terminates the program.

We must also register the **signal_handler** for the **SIGINT** signal. This signal is sent to all processes in the foreground process group when the user enters the interrupt character (that is Ctrl-C).

Note that for the sake of brevity, error checking is omitted in this code.

The Bluetooth client running on the BeagleBone Black has the following source code (**Listing 43)**.

Listing 43.

```
import bluetooth
import time

serverMACAddress = '00:15:83:15:A3:10'
port = 6
cnt = 0
while cnt < 10:
    s = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
    s.connect((serverMACAddress, port))
    s.send("Test message from the BBB Client to the Linux Mint Server.")
    s.close()
    cnt = cnt + 1
    time.sleep(4)
```

The client application is much simpler then the server. Here we need to create the Bluetooth socket and establish the connection to the Bluetooth server. That is accomplished by the following statements:

```
s = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
s.connect((serverMACAddress, port))
```

Then the client sends the text string to the server:

```
s.send("Test message from the BBB Client to the Linux Mint Server.")
```

When done, the program closes the socket:

s.close()

The client repeats those operations every 4 s in the **while cnt < 10** loop until the counter **cnt** remains less than 10.

The Bluetooth server output is shown below:

```
root@beaglebone:/# python SimpleBTServer.py
Waiting for incoming requests from Bluetooth clients on port 6…
Press Ctrl+C for exit…
Test message from the BBB Client to the Linux Mint Server.
Test message from the BBB Client to the Linux Mint Server.
Test message from the BBB Client to the Linux Mint Server.
Test message from the BBB Client to the Linux Mint Server.
Test message from the BBB Client to the Linux Mint Server.
Test message from the BBB Client to the Linux Mint Server.
Test message from the BBB Client to the Linux Mint Server.
Test message from the BBB Client to the Linux Mint Server.
Test message from the BBB Client to the Linux Mint Server.
Test message from the BBB Client to the Linux Mint Server.
Test message from the BBB Client to the Linux Mint Server.
Test message from the BBB Client to the Linux Mint Server.
^CTerminating on Ctrl+C…
```

The following project illustrates the design of a wireless measurement system where the BeagleBone Black will transfer the value of the analog signal measured on the analog input **AIN0**. The circuit diagram of such a system is shown in **Fig.90**.

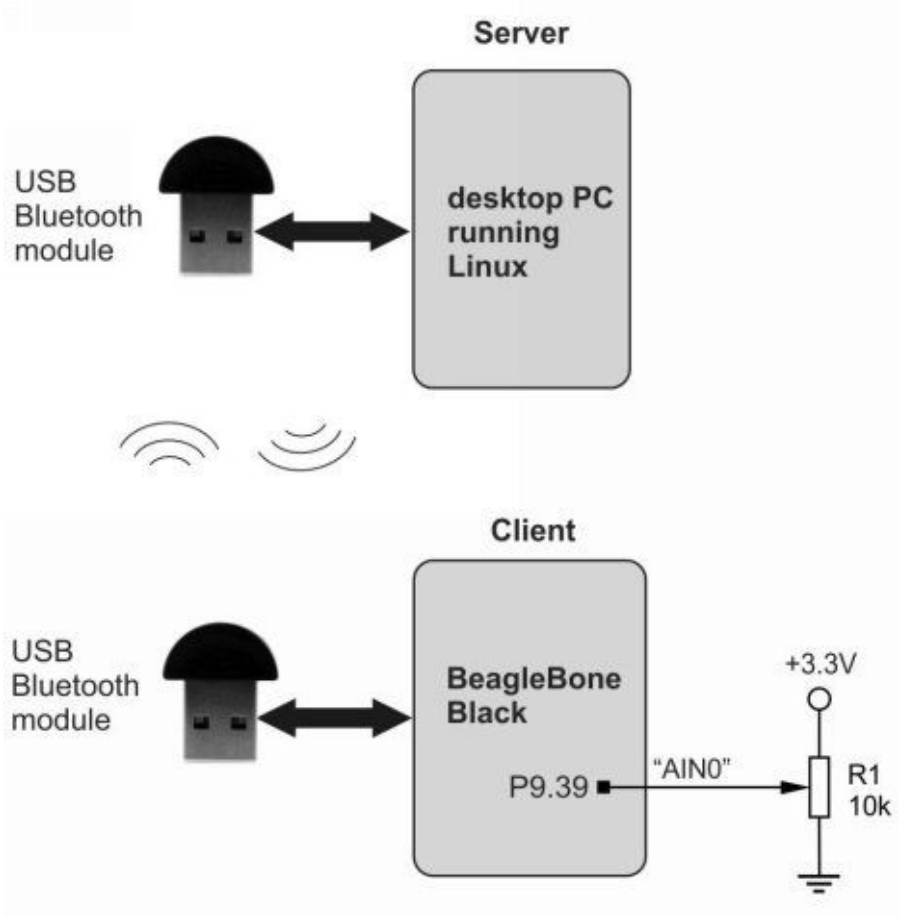


Fig.90

In this system, the analog input signal is simulated by the potentiometer R1 whose wiper is connected to pin **P9.39** (channel **AIN0**) of the BeagleBone Black. **Remember that the amplitude of an input signal applied to any analog input of the BeagleBone Black must not exceed 1.8V!**

The Python source code of the Bluetooth server running in Linux Mint is the same as that shown in **Listing 42**.

The source code for the Bluetooth client is shown in **Listing 44**.

Listing 44.

```
import Adafruit_BBIO.ADC as ADC
import signal
import sys
import bluetooth
import time
```

```
def signal_handler(signal, frame):
    print "Client is terminating on Ctrl+C…"
    s.close()
    sys.exit(0)

ADC.setup()
CH0 = "P9_39"
serverMACAddress = '00:15:83:15:A3:10'
port = 6

signal.signal(signal.SIGINT, signal_handler)
while 1:
    inpVal = ADC.read(CH0) * 1.8
    s = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
    s.connect((serverMACAddress,port))
    s.send("Input voltage on CH0: " + str(round(inpVal, 2)) + "V")
    s.close()
    time.sleep(5)
```

BeagleBone Black and Arduino Due in measurement systems using Bluetooth

Wireless systems with a Bluetooth interface can use development boards capable of low-level signal processing. Arduino Uno, Arduino Due and other popular modules can be fit into wireless systems using a Bluetooth interface. Many Bluetooth devices support a serial interface, so an Arduino board can interact with a Bluetooth device through the standard serial lines (TXD and RXD).

The following material is dedicated to use of a popular Arduino Due board in wireless systems with the BeagleBone Black. To communicate with the BeagleBone we will employ a HC-06 Bluetooth module connected to Arduino Due via a serial interface.

The block diagram of a wireless measurement system using the BeagleBone Black and Arduino Due may look like the following (**Fig.91**).

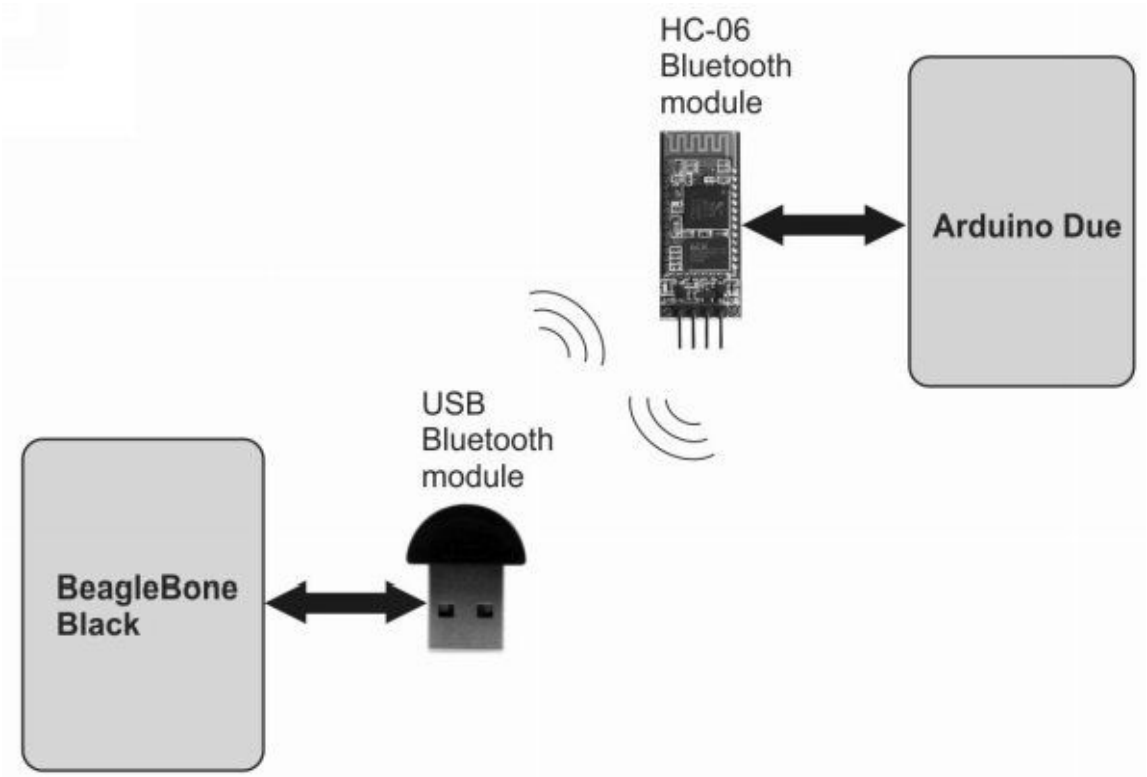


Fig.91

In this system, the BeagleBone Black can receive the data processed by Arduino Due or send commands to Arduino Due. As you can see, the BeagleBone uses an ordinary USB Bluetooth device for wireless communication with the serial Bluetooth device HC-06

connected to Arduino Due.

Note that you can also apply other Bluetooth modules with a serial interface, but remember that those modules will have to provide 3.3V signal levels on their TXD and RXD pins!

The HC-06 module looks like the following (**Fig.92**).

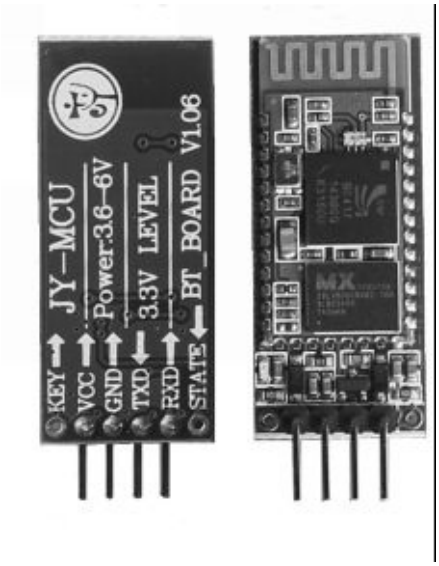


Fig.92

The connections between the HC-06 Bluetooth device and Arduino Due are shown in **Fig.93**.

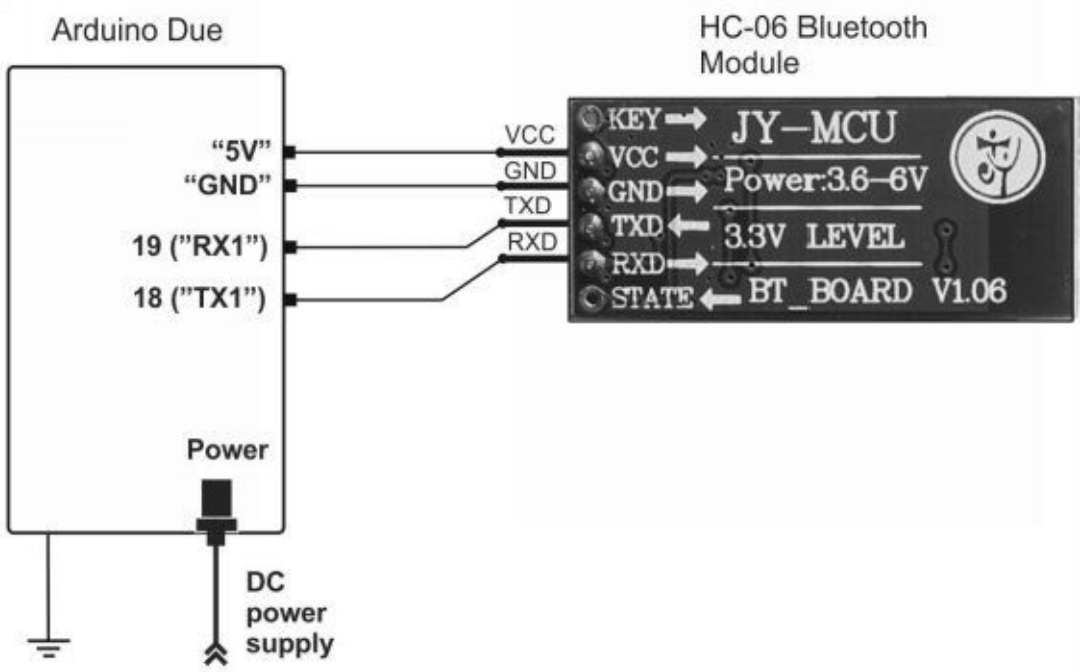


Fig.93

In this circuit, the "**TXD**" pin of the HC-06 Bluetooth module is connected to pin **19**

(“**RX1**”) of Arduino Due; the “**RXD**” pin of HC-06 is connected to pin **18** (“**TX1**”) of Arduino Due. The power pin **VCC** of HC-06 is connected to pin “**5V**” of Arduino Due. The common wire “**GND**” of HC-06 is tied to pin “**GND**” of Arduino Due. Note that high level of signals on pins **TXD** and **RXD** of the HC-06 Bluetooth module are 3.3V.

The pin connections are summarized in the table below.

Arduino Due pin	**HC-06 pin**
19(“RX1”)	TXD
18(“TX1”)	RXD
5V	VCC
GND	GND

This is a basic configuration which will be used in the projects that follow.

In order to develop wireless applications, we have to install the Bluetooth software packages on the BeagleBone Black. The installation of the Bluetooth program interface on the BeagleBone described in detail in numerous sources, so we will discuss only the key commands which may come in handy when developing applications using Bluetooth.

Before starting off, we need to know if Bluetooth devices are present in Linux by entering the command:

```
root@beaglebone:/# hcitool dev
Devices:
    hci0        00:15:83:3D:0A:57
```

To find out remote Bluetooth devices we should enter:

```
root@beaglebone:/# hcitool scan
Scanning …
    98:D3:31:B3:A4:DF            HC-06
    9C:02:98:1E:73:6B            S5610
```

It is seen that two Bluetooth devices can be reached from our BeagleBone Black. In our case, we need to establish the communication between the local device hci0 (BeagleBone Black) and the remote device HC-06 (Arduino Due). To do that we can use the **rfcomm** command.

rfcomm command is used to set up, maintain, and inspect the RFCOMM configuration of the Bluetooth subsystem in the Linux kernel. If no command is given, or if the option **-a** is used, **rfcomm** prints information about the configured RFCOMM devices.

This command has the following syntax:

```
root@beaglebone:/# rfcomm help
RFCOMM configuration utility ver 4.99
Usage:
    rfcomm [options] <command> <dev>

Options:
    -i [hciX|bdaddr]      Local HCI device or BD Address
    -h, —help             Display help
    -r, —raw              Switch TTY into raw mode
    -A, —auth             Enable authentication
    -E, —encrypt          Enable encryption
    -S, —secure           Secure connection
    -M, —master           Become the master of a piconet
    -f, —config [file]    Specify alternate config file
    -a                    Show all devices (default)

Commands:
    bind     <dev> <bdaddr> [channel]     Bind device
    release  <dev>                        Release device
    show     <dev>                        Show device
    connect  <dev> <bdaddr> [channel]     Connect device
    listen   <dev> [channel [cmd]]        Listen
```

watch <dev> [channel [cmd]] Watch

To establish the connection between hci0 and HC-06 we should enter:

root@beaglebone:/# rfcomm bind /dev/rfcomm0 98:D3:31:B3:A4:DF 1

If **rfcomm** succeeds, the communication channel 1 between both devices has been established. The data will be transferred using the character device /dev/rfcomm0 which has been created by the **rfcomm** command:

root@beaglebone:/# ls -l /dev/rfcomm*

crw-rw–T 1 root dialout 216, 0 Dec 26 17:50 /dev/rfcomm0

We can check the state of the communication channel by entering **rfcomm** without parameters:

root@beaglebone:/# rfcomm

rfcomm0: 98:D3:31:B3:A4:DF channel 1 connected [tty-attached]

We can also check the active connections by entering:

root@beaglebone:/# hcitool con

Connections:

< ACL 98:D3:31:B3:A4:DF handle 42 state 1 lm MASTER

The next command allows to disconnect the HC-06 device:

root@beaglebone:/# hcitool dc 98:D3:31:B3:A4:DF

The above command comes in handy when we need to release device or reestablish the connection to the remote Bluetooth device when the following error occurs:

IOError: [Errno 16] Device or resource busy: ‘/dev/rfcomm0’

Once the communication between Bluetooth devices has been established, we can test the data flow through the communication channel.

Enter the following command in the terminal window on the BeagleBone Black:

root@beaglebone:/# echo “BBB to Arduino Test String” > /dev/rfcomm0

This string will be received by the remote system (Arduino Due).

Again, if we transfer data from Arduino Due to the remote BeagleBone Black, we can check this by entering the following command in the terminal window on the BeagleBone Black:

root@beaglebone:/# cat /dev/rfcomm0

DUE-BBB Test RXD

DUE-BBB Test RXD

DUE-BBB Test RXD

DUE-BBB Test RXD

DUE-BBB Test RXD

DUE-BBB Test RXD

DUE-BBB Test RXD

…

Here the **cat** command reads the device /dev/rfcomm0 and outputs the text string received from Arduino Due board.

The **rfcomm** command discussed in this section will be used in Python applications that will be described in the following sections.

BeagleBone Black and Arduino Due in measurement systems using Bluetooth: project 1

This project illustrates wireless control of the PWM signal issued by Arduino Due. The command will be transferred wirelessly from the BeagleBone Black using Bluetooth. The circuit diagram of the project is shown in **Fig.94**.

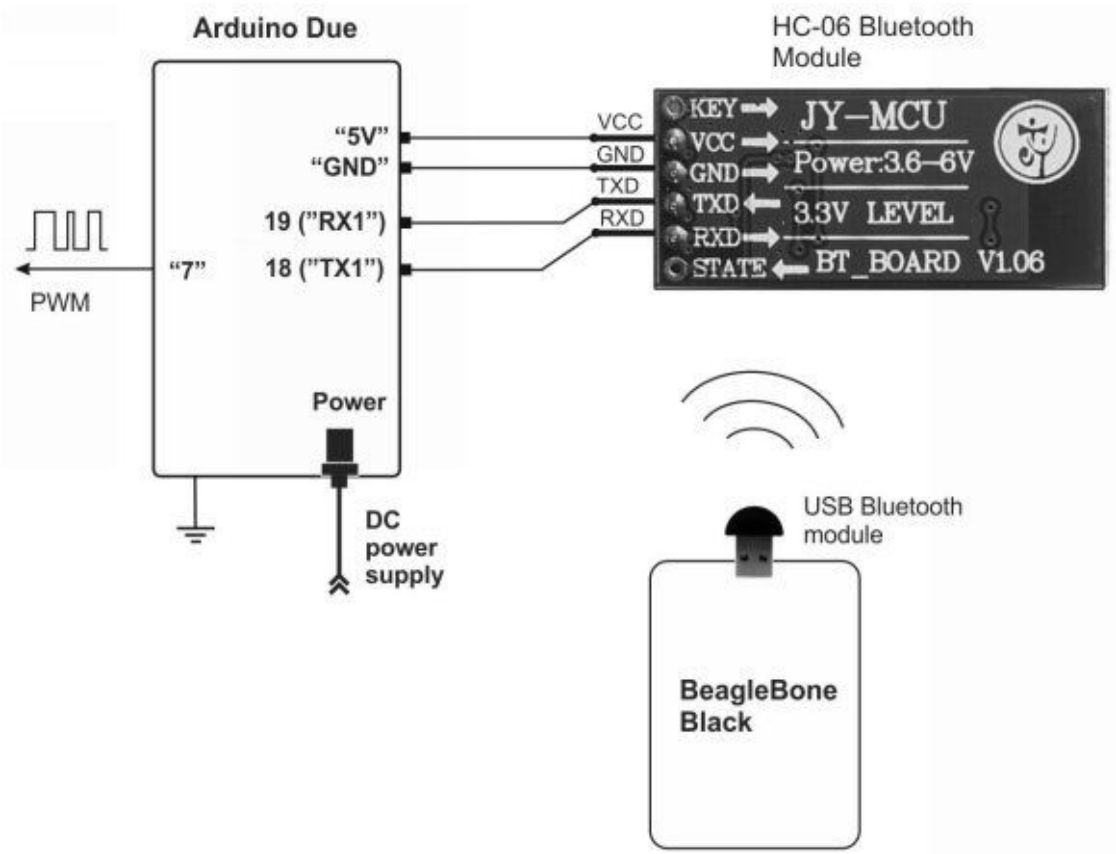


Fig.94

In this system, the HC-06 Bluetooth module receives the command from the BeagleBone Black; Arduino Due reads the command through the serial interface, then parses the command string and adjusts the parameters of the PWM signal on pin "**7**" accordingly.

The BeagleBone Black Python source code is shown in **Listing 45**.

Listing 45.

```
import subprocess
import os
```

```
import time
```



```
if not os.path.isfile(“/dev/rfcomm1”):
    subprocess.call([“rfcomm”, “bind”, “/dev/rfcomm1”, “98:D3:31:B3:A4:DF”, “1”])
    time.sleep(0.5)
f1 = open(“/dev/rfcomm1”, “w”)
f1.write(“PWM 800 600\n”) # PWM freq = 4000000/800 Hz, Duty = 600/800
f1.close()
time.sleep(0.5)
subprocess.call([“rfcomm”, “release”, “/dev/rfcomm1”])
subprocess.call([“rm”, “/dev/rfcomm1”])
```

This application uses the rfcomm command to establish the communication channel to the remote Bluetooth device. This is done by the following statement:

```
subprocess.call([“rfcomm”, “bind”, “/dev/rfcomm1”, “98:D3:31:B3:A4:DF”, “1”])
```

When succeded, the statement creates the communication channel 1 to the remote device; the character device /dev/rfcomm1 will be used for data transfer.

Before creating the the communication channel it is worth checking if the device /dev/rfcomm1 already exists by executing:

```
if not os.path.isfile(“/dev/rfcomm1”)
```

The statement

```
time.sleep(0.5)
```

provides the delay of 0.5 s (you can select another value) needed to complete the previous operation.

Then the command string is transferred to the remote device using common file I/O functions:

```
f1 = open(“/dev/rfcomm1”, “w”)
f1.write(“PWM 800 600\n”) # PWM freq = 4000000/800 Hz, Duty = 600/800
f1.close()
```

That is followed by a delay (time.sleep(0.5)) which is needed to flush I/O buffers. Then the program code closes the communication channel to the remote device and explicitly removes the /dev/rfcomm1 device. That is done by the following sequence:

```
subprocess.call([“rfcomm”, “release”, “/dev/rfcomm1”])
subprocess.call([“rm”, “/dev/rfcomm1”])
```

The Arduino Due source code is shown in **Listing 46**.

Listing 46.

```
#define PWM_CLOCK 4000000 // Clock frequency for PWM = 4.0MHz
#define CH  6             // PWM channel 6 (pin 7 of DUE) is selected

int PWM_PERIOD = 400; // the initial frequency is set to 10.0KHz (PWM_CLOCK / PWM_PERIOD)
int PWM_DUTY = 200;    // the initial duty = 50%

String rcvSerial;
String cmd;    // the received command is saved here
String sPeriod; // the period substring goes here
String sDuty;   // the duty substring goes here (0-255)

int period; // an actual period
int duty;   // an actual duty
```

```
void setup() {
  // put your setup code here, to run once
pinMode(7, OUTPUT); // PWML 6 is selected
analogWrite(7, 100);

PWMC_DisableChannel(PWM, CH);
PWMC_ConfigureClocks(PWM_CLOCK, 0, VARIANT_MCK); // Clock A is set to 4.0MHz, B = 0
PWMC_ConfigureChannel(PWM, CH,PWM_CMR_CPRE_CLKA,0,0);
PWMC_SetPeriod (PWM, CH, PWM_PERIOD);
PWMC_SetDutyCycle(PWM, CH, PWM_DUTY);
PWMC_EnableChannel(PWM,CH);

Serial1.begin(9600);
}

void parseStr(String s1)
{
int spaceIndex = s1.indexOf(' ');
cmd = s1.substring(0, spaceIndex);
cmd.trim();

if (cmd.equalsIgnoreCase("pwm"))
{
  int spaceIndex2 = s1.indexOf(' ', spaceIndex+1);
  sPeriod = s1.substring(spaceIndex, spaceIndex2);
  sPeriod.trim();
  period = sPeriod.toInt();

  int spaceIndex3 = s1.indexOf(' ', spaceIndex+2);
  sDuty = s1.substring(spaceIndex3);
```

```
  sDuty.trim();
  duty = sDuty.toInt();
}
}

void loop() {
  // put your main code here, to run repeatedly:
  if(Serial1.available() > 0)
  {
    rcvSerial = Serial1.readStringUntil('\r');
    parseStr(rcvSerial);
    if (duty > 0 && duty < 1000)
    {
    PWM_PERIOD = period;
    PWM_DUTY = duty;

    PWMC_DisableChannel(PWM, CH);
    PWMC_SetPeriod (PWM, CH, PWM_PERIOD);
    PWMC_SetDutyCycle(PWM, CH, PWM_DUTY);
    PWMC_EnableChannel(PWM,CH);
    }
  }
  delay(3000);
}
```

BeagleBone Black and Arduino Due in measurement systems using Bluetooth: project 2

This project illustrates passing the result of A/D conversion from Arduino Due to the remote BeagleBone Black. The circuit diagram of the system is shown in **Fig.95**.

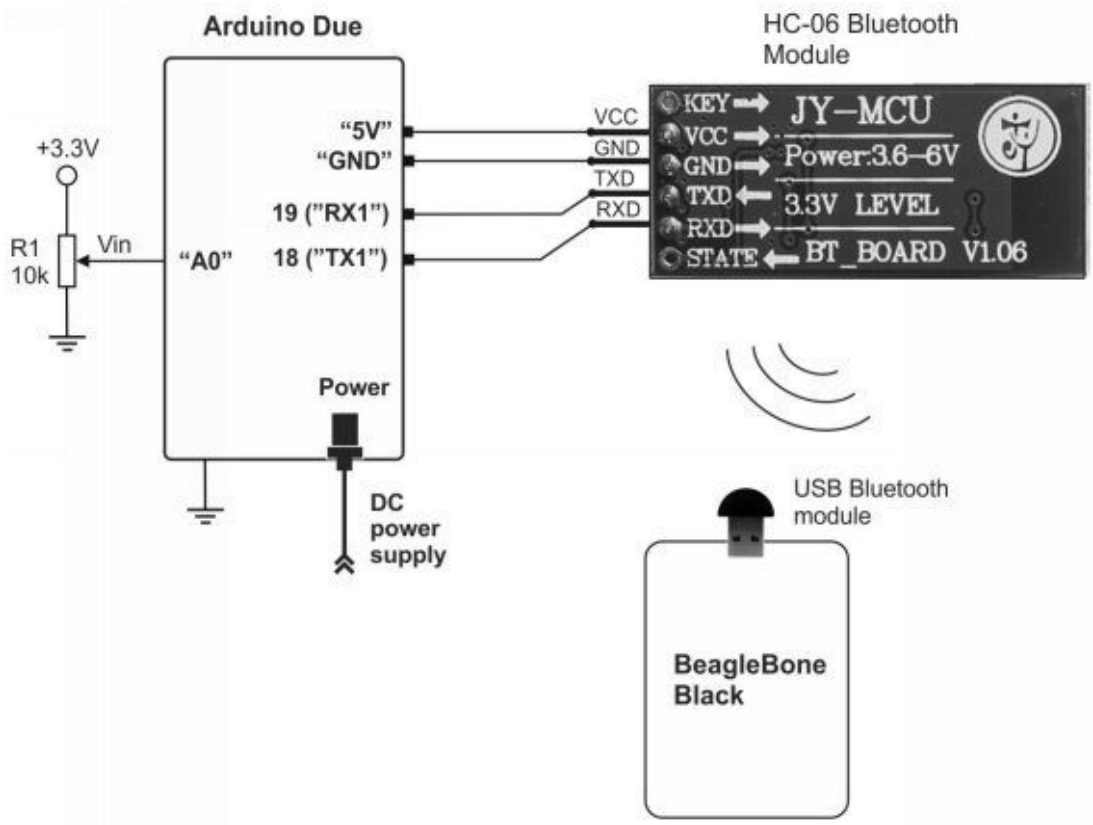


Fig.95

Here the analog input voltage taken from the wiper of a potentiometer R1 is fed to to the analog channel A0 of Arduino Due. The result of A/D conversion is then transferred through the serial interface to the HC-06 Bluetooth device.

The BeagleBone Black Python source code is shown in **Listing 47**.

Listing 47.

```
import subprocess
import os
```

```
import time

if not os.path.isfile("/dev/rfcomm1"):
    subprocess.call(["rfcomm", "bind", "/dev/rfcomm1", "98:D3:31:B3:A4:DF", "1"])
    time.sleep(0.5)
f1 = open("/dev/rfcomm1", "r")
for i in range (1, 21):
    print (f1.readline())
f1.close()
subprocess.call(["rfcomm", "release", "/dev/rfcomm1"])
subprocess.call(["rm", "/dev/rfcomm1"])
```

This code is similar to that shown in **Listing 45**, except that the device /dev/rfcomm1 is opened for read operation by the statement:

```
f1 = open("/dev/rfcomm1", "r")
```

The data are read and output in the **fo**r loop which provides 20 iterations. The rest is the same as in **Listing 45**.

The Arduino Due source code is shown in **Listing 48**.

Listing 48.

```
int CH0 = A0;       // potentiometer wiper (middle terminal) connected to pin A0
                    // outside leads are connected to ground and +3.3V
int inpVal = 0;     // variable to store the binary code of a value read
float VREF = 3.3; // reference voltage to the A/D Converter
float res;

int cnt = 0;
```

```
void setup() {
  Serial1.begin(9600);
    // change the resolution to 12 bits and read A0
  analogReadResolution(12);
}

void loop() {
    inpVal = analogRead(CH0);
    res = VREF/4096 * (float)inpVal;
    if (cnt > 100)
    cnt = 0;
    Serial1.print(cnt++);
    Serial1.print(“: Channel A0 input voltage: “);
    Serial1.print(res,3);
    Serial1.println(” V”);
    delay(5000);
}
```